НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»

Теплоенергетичний факультет

Кафедра автоматизації проектування енергетичних процесів і систем

До захисту допущено:

Завідувач кафедри

_____ Олександр Коваль

«___»_____2020 р.

Дипломна робота

на здобуття ступеня бакалавра
за освітньо-професійною програмою «Програмне забезпечення розподілених систем»
спеціальності 121 «Інженерія програмного забезпечення»
на тему: «Інструментальні засоби розробки мікросервісів на основі контрактного підходу»

Виконав:
студент IV курсу, групи ТВ-61
Калашников-Травін Владислав Володимирович _____

Керівник:
доцент, к.ф.-м.н.
Тарнавський Юрій Адамович _____

Рецензент:

_____

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ - 2020

Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”

Факультет теплоенергетичний

Кафедра автоматизації проектування енергетичних процесів і систем

Рівень вищої освіти перший рівень

Напрям підготовки: 121 Інженерія програмного забезпечення

Спеціалізація: Програмне забезпечення розподілених систем

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Олександр Коваль
(підпис)
”___”_____2020р.

ЗАВДАННЯ

на дипломну роботу студенту

Калашникову-Травіну Владиславу Володимировичу
(прізвище, ім’я, по батькові)

1. Тема роботи Інструментальні засоби розробки мікросервісів на основі контрактного підходу
керівник роботи Тарнавський Юрій Адамович, доцент, кандидат наук
(прізвище, ім’я, по батькові науковий ступінь, вчене звання)

затверджена наказом вищого навчального закладу від ”25” травня 2020р. № 1168-с

2. Строк подання студентом роботи _____
3. Вихідні дані до роботи мова програмування C# 8.0, програмні платформи .NET Core 3.1 та .NET Standard 2.1, середовище розробки JetBrains Rider 2020.1.3, брокер повідомлень RabbitMQ 3.8.3
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) проаналізувати існуючі підходи до розробки розподіленого програмного забезпечення, розробити набір бібліотечних пакетів, що створюють каркас для побудови розподілених продуктів та розв’язують інфраструктурні проблеми прозорої комунікації компонентів розподіленої системи між собою.
5. Перелік ілюстративного матеріалу 1 — Титульний слайд; 2 — Інфраструктурна

задача комунікації мікросервісів; 3 — Мета створення; 4 — Принцип CQRS та типи контрактів; 5 — Діаграма пакетів фреймворку “Standalone”; 6 — Конвеєр обробки контрактів; 7 — Діаграма послідовності обробки запитів; 8 — Використані технології та системні вимоги; 9 — Встановлення у користувацький додаток; 10 — Рекомендована структура користувацького додатку; 11 — Навантажувальний тест шин обробки; 12 — Стрес-тест шин обробки; 13 — Приклад спільної роботи мікросервісів демонстраційного додатку; 14 — Висновки.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання ”11” жовтня 2019 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітки
1.	Затвердження теми роботи	08.03 — 14.03	
2.	Вивчення та аналіз задачі	14.04 — 21.04	
3.	Розробка архітектури та загальної структури	21.04 — 27.04	
4.	Розробка структур окремих підсистем	27.04 — 06.05	
5.	Програмна реалізація системи	06.05 — 20.05	
6.	Оформлення пояснювальної записки	20.05 — 08.06	
7.	Захист програмного продукту	08.06	
8.	Передзахист	08.06	
9.	Захист	15.06	

Студент _____ Калашников-Травін В. В.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Тарнавський Ю. А.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Зі збільшенням об’єму продуктів програмного забезпечення, розробники все частіше постають перед розв’язанням проблеми побудови великих розподілених систем та їх підтримки. Розподілені системи вимагають додаткових зусиль задля створення нового чи змінення вже існуючого функціоналу через додаткові накладення на комунікації між компонентами фрагментованої системи.

Метою даної роботи є розробити базове бібліотечне програмне забезпечення, що розв’язує інфраструктурну проблему зв’язку між компонентами (мікросервісами) розподілених систем та дозволяє розробникам розподіляти та використовувати функціонал створюваного додатку прозоро, незалежно від його фактичного місцезнаходження у кодовій базі рішення.

Записка цієї роботи містить 60 сторінок, 20 рисунків, 20 посилань та 3 додатки.

Ключові слова до роботи: CQRS, розробка розподілених систем, мікросервісна архітектура, програмні контракти, пересилання повідомлень, віддалене виконання, прозоре виконання.

ABSTRACT

As software products grow, developers are increasingly facing the problem of building and maintaining large distributed systems. Distributed systems require additional effort to create new or modify existing functionality through additional overlays on communications between components of a fragmented system.

The purpose of this work is to develop basic library software that solves the infrastructural problem of communication between components (microservices) of distributed systems and allows developers to distribute and use the functionality of the created application transparently, regardless of its actual location in the code base of the solution.

The following sheet contains 60 pages, 20 images, 20 references and 3 additions.

Keywords list: CQRS, distributed systems development, microservice architecture, program contracts, message transporting, remote execution, transparent execution.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

1. POCO (з англ. — Plain Old CLR Object), POJO (з англ. — Plain Old Java Object) — плоский клас, що має властивості та поля, але не має методів;
2. Git — найпоширеніша система контролю версій коду;
3. CQRS (з англ. — Command Query Responsibility Segregation) — архітектурний принцип, за яким слід розділяти обробку запитів та команд;
4. Мікросервіс — окремий процес, що являє собою частину розподіленого додатку та відповідає за реалізацію певного набору вимог до додатку;
5. DLL (з англ. — Dynamic Load Library) — програмний модуль, що може бути завантажений під час роботи додатку;
6. CLR (з англ. — Common Language Runtime) — оточення виконання програм, що написані мовами сімейства .NET;
7. IDE (з англ. — Integrated Development Environment) — програмне забезпечення, що надає засоби для розробки та налагодження додатків;
8. API (з англ. — Application Programming Interface) — чітко визначений набір програмних інтерфейсів для взаємодії компонентів, модулів, додатків тощо.

ВСТУП

Будь-який програмний комплекс створений для розв’язання деякої конкретної задачі або набору задач. У часи початку розвитку програмних рішень як способу поліпшення вирішення задач, програми представляли собою невеликий набір вихідних кодів та їх конфігурацій. Розробка додатків такого розміру зазвичай не потребувала великих команд розробки, а вимоги до експлуатаційного обладнання були досить малі в порівнянні з сьогоденням. З плином часу, задачі ускладнювалися та врізноманітнювалися, програми набирали розмірів та навантаження на сервери збільшувалось. Крім того, такі системи переважно комерційні, а отже мають приносити прибуток. У реальному світі, чим більш прибутковий додаток, тим більше можливостей він надає кінцевому користувачу, а чим більше можливостей, тим більше програмного коду необхідно створювати, підтримувати та розгортати на виробничому середовищі. З ростом вимог до програмного забезпечення, росли і розміри команд розробки, обсяг засобів розробки, призначених для різних класів задач та потреби в потужному апаратному забезпеченні. В контексті розробки великих та високо навантажених програмних комплексів, монолітна архітектура додатку, як правило, перетворюється зі свого чистого стану у нагромадження неструктурованих рішень, що надзвичайно важко підтримувати. Через те що компоненти такої системи розвиваються разом, їх також необхідно змінювати разом. Такі перетворення уповільнюють процес розробки: кожну наступну частину функціоналу системи буде складніше розвивати. Наведені вище причини призвели до необхідності в розподіленні систем на менші компоненти. Це значно спрощує розробку громіздких систем, надаючи наступні можливості [1]:

— розробляти та розгортати компоненти системи незалежно одне від одного;
— розробляти компоненти декількома командами розробки, що пришвидшує появу нових можливостей системи;
— збільшення надійності системи; якщо один з компонентів вийде з ладу, то інші продовжать свою роботу;
— масштабування компонентів системи за необхідності можливо підняти продуктивність окремих компонентів системи, збільшуючи кількість запущених компонентів та розподіляючи між ними навантаження.

Наведені вище жорсткі вимоги призвели до розвитку мікросервісної архітектури. За своїм призначенням, вона дуже схожа на unix-подібні програми — кожен мікросервіс виконує малий обсяг роботи, проте виконує його справно та ефективно. Це ставить перед розробниками задачу — поєднати сервіси спільною комунікаційною шиною задля забезпечення отримання та оновлення інформації між ними. Очевидно, що в світі таких систем багато і розв’язувати цю задачу окремо під кожну досить нелогічно і затратно, адже можливо виділити частини системи у бібліотеки та використовувати їх повторно для побудови інших програмних продуктів.

Актуальність теми дипломної роботи полягає в тому, що з ростом складності програмних систем необхідна принципово нова організація архітектури програмних продуктів. Існує потреба в заміні застарілої та мало ефективної монолітної архітектури додатків, яка значно сповільнює розвиток та ускладнює розширення громіздких програмних комплексів, що в свою чергу підвищує вартість розробки та складність у підтримці таких комплексів. При цьому, альтернатива не повинна ускладнювати розробку в плані турбування про реалізацію комунікації компонентів

систем, а навпаки, надавати розробникам прозоре рішення для побудови мікросервісного додатку.

Метою даної роботи є вирішення інфраструктурної проблеми комунікації мікросервісів між собою. Результатом роботи має бути програмне забезпечення у вигляді бібліотек, що призначене для спрощення побудови високоскладних систем та їх розподілення шляхом застосування мікросервісної архітектури та архітектурного шаблону CQRS. Ці бібліотеки мають поставлятися у вигляді пакетів, що можуть бути використані для побудови багатосервісного enterprise-рішення. Пропонується розглянути та розробити інструменти комунікації мікросервісів на основі контрактів між мікросервісами — поширених класів, що представляють собою повідомлення, якими обмінюються компоненти програмного комплексу. Такий підхід надає можливість реалізувати прозорий обмін повідомленнями, що позитивно впливає на швидкість розробки та простоти системи, навіть за умови її великих розмірів.

Результуючий набір пакетів може бути використаний розробником або командою розробки для створення фрагментованого на мікросервіси додатку. Крім того, цей набір може бути використаний без використання транспорту загалом, що дозволяє розробляти невеликі монолітні додатки з перспективою розбиття на мікросервіси.

1. ІНФРАСТРУКТУРНА ЗАДАЧА КОМУНІКАЦІЇ МІКРОСЕРВІСІВ

Фреймворк — зовнішній бібліотека або набір бібліотек з готовим програмним кодом, що задає початкову структуру програмної системи, спрощує розробку та об’єднання її програмних компонентів. Фреймворк є каркасом для продукту, що його використовує та не залежить від кожного конкретного продукту. Фреймворк розвивається окремо від будь-якого продукту та вважається окремим проектом.

Результатом даної роботи має бути фреймворк “Standalone”, що задає архітектурний каркас для створення мікросервісного додатку на базі архітектурного принципу CQRS та дозволяє розробникам сконцентруватися на розв’язанні задач бізнес-логіки і не піклуватися про інфраструктурні проблеми комунікації мікросервісів між собою. Основоположним для побудови розподілених систем на базі результуючого фреймворку має стати контрактний підхід, за яким мікросервіси мають спільні визначення контрактів, за допомогою яких відбувається звернення одного мікросервісу до іншого. Такий підхід позитивно впливає на типобезпеку створення та обміну повідомленнями, а також задає чіткий програмний інтерфейс для кожного мікросервісу.

Більшість розподілених систем базується на явному обміні повідомленнями між процесами, тобто віддалені операції виклику і обробки повідомлень безпосередньо використовують рівні зв'язку. Це призводить до непрозорого доступу до функціональності всередині розподіленої системи, а отже і до підвищення її складності в цілому [2]. З метою поліпшити рівень прозорості таких систем, особливістю реалізації даного рішення має бути повна прозорість виконання, що

надаватиме розробникам можливість не піклуватися про те, де саме і як виконується метод [3], адже знаходження та обробку контрактів бере на себе фреймворк “Standalone”. Реалізація передачі повідомлень є комплексом робіт, що включає в себе інтеграцію з брокером повідомлень, підтримку вибору та заміни засобів серіалізації/десеріалізації повідомлень та відповідей, обробка тайм-ауту очікування відповіді про обробку тощо.

У відкритому доступі вже існує перелік бібліотек, що виконують схожі задачі. Пропонується проаналізувати подібні фреймворки, наслідувати їх переваги та позбутися можливих недоліків. Так, наприклад, абсолютна більшість аналогічних рішень має єдину програмну бібліотеку, що містить у собі всі реалізації, що негативно впливає на розміри та швидкість завантаження таких бібліотек під час запуску додатка. Поставка фреймворку “Standalone” має здійснюватися шляхом встановлення тільки необхідних окремих програмних пакетів, окремо у кожен з сервісів додатку. Кожен модуль фреймворку має бути розділений на бібліотечні пакети невеликого розміру, що мають обмежений набір функціонал та можуть бути інтегровані одне з одним.

Задля створення точок розширення, пропонується розробити реалізацію за принципом конвеєрної обробки. Це дозволяє створювати новий функціонал та розширення вже існуючого, не змінюючи основну кодову базу фреймворку. Крім того, конвеєр має підтримувати користувацькі розширення, тобто вживлення додаткового функціоналу розробниками кінцевих додатків, без внесення будь-яких змін у код фреймворку.

2. ВИКОРИСТАННЯ ПРИНЦИПУ CQRS ТА ВІДДАЛЕНОЇ ОБРОБКИ ПРИ ПОБУДОВІ МІКРОСЕРВІСНИХ ДОДАТКІВ

2.1 Поняття мікросервісної архітектури та CQRS

Мікросервісна архітектура — принцип організації розподіленої системи на основі мікросервісів та їх взаємодії між собою та з зовнішнім програмними системами, а також створення направляючих стилів написання та імплементації кінцевої системи, її розвитку та подальшої підтримки. Єдиний додаток будується як набір невеликих сервісів, кожен з яких працює у власному процесі і комунікує з іншими, використовуючи мережеві засоби, наприклад HTTP та AMPQ протоколи [4]. Ці сервіси побудовані навколо бізнес-потреб і розгортаються незалежно з використанням повністю автоматизованого середовища, наприклад GitLabCI, TravisCI або Jenkins [5].

Мікросервіс — програмна одиниця розподілення функціоналу, що містить у собі бізнес-логіку, що обмежена певним контекстом. Наприклад, мікросервіс авторизації/аутентифікації має обмежений контекст, що включає в себе лише функціонал, що відповідає за вхід користувача в систему: збереження та перевірка даних входу користувачів, видача токенів авторизації або сесії та їх валідація, відновлення паролю тощо.

Мікросервіс можна описати наступними характеристиками:

— невеликий — містить лише той функціонал, що відповідає обмеженому контексту;
— незалежний — розгортання та обслуговування відбувається незалежно від інших сервісів;
— розвиток мікросервісу залежить від потреб бізнесу та як наслідок — розширення обмеженого контексту;
— створення нових мікросервісів залежить від утворення нових вимог до додатку та як наслідок — створення нових обмежених контекстів, що в цілому і представляють всю систему, що розробляється.

Реалізація у даній роботі базується на принципі проектування CQRS — Command Query Responsibility Segregation. Принцип CQRS був винайдений Бертраном Мейером під час створення мови програмування Eiffel [6]. Основною ідеєю принципу є розділення отримання стану системи та зміни цього стану. В загальному випадку, принцип накладає обмеження на методи в системі: метод має бути або командою (command), що змінює дані в системі, або запитом (query), що повертає дані, але не одночасно [7]. Фреймворк “Standalone” пропонує користувачеві виконувати команди і запити прозоро, незважаючи на місце знаходження відповідного обробника (локальний або віддалений).

Раніше CQRS принцип використовували частіше за все у контрактному програмуванні, оскільки запити ніколи не змінюють стан системи, а отже їх можна викликати з тверджень. На сьогоднішній день домінуючою парадигмою розробки програмного забезпечення є об’єктно-орієнтоване програмування, у якому CQRS принцип також успішно застосовується [7].

На сьогоднішній день існує безліч модифікацій цього принципу. Так, наприклад, використовуючи CQRS можна розподілити дані на два окремих сховища

даних: для запису і для зчитування. У звичайному випадку, додаток має одну базу даних, що може одночасно обробляти запити на отримання, створення, оновлення чи видалення даних. У високонавантажених системах така база даних може стати найбільш слабкою ланкою системи, тобто компоненти системи будуть вимушені очікувати, доки база даних не завершить обробку запиту, що в свою чергу сповільнює загальну роботу системи. Ця проблема вирішується вертикальним масштабуванням, тобто простим оновленням апаратного забезпечення серверу бази даних. Проте, врешті решт таке оновлення може бути досить коштовним, особливо у періоди простою системи. Принцип CQRS дозволяє вирішити цю проблему за допомогою розділення одно сховища на два, при цьому не обов’язково однакові [8]. Сховище для запису обробляє лише запити на оновлення даних, тож навантаження на таку базу даних буде значно менше. Сховище для зчитування може бути денормалізованою базою даних, що значно пришвидшує матеріалізацію та отримання даних за запитом. Такий підхід також значно зменшує навантаження на апаратну частину, що в цілому є більш ефективним, ніж збирати необхідні дані з декількох таблиць, поєднувати їх і лише потім матеріалізувати та віддати клієнту. Крім цього, замість реальної бази даних для зчитування можливо застосувати швидкодіючий кеш, що позбавляє необхідності робити однакові запити до серверу бази даних. Оскільки база даних тримає всі дані на диску, зчитування з кешу завжди буде швидшим, оскільки кеш зберігає дані в оперативній пам’яті комп’ютера, яка в рази швидше за звичайне дискове сховище.

2.2 Порівняння монолітної та мікросервісної архітектур

Додаток, побудований з використанням монолітної архітектури додаток являє собою додаток, що доставляється на оточення через єдине розгортання. Таким є додаток, розгорнутий на сервері у вигляді однієї .NET DLL збірки, Node.JS додаток з однією точкою входу або PHP скрипт, оброблюючий клієнтські запити. Як видно з прикладів, головною відмінністю монолітного додатку від мікросервісного являється єдиний процес, що містить у собі всі методи та функції, що так чи інакше обробляють вхідні запити. На рисунках 2.1 та 2.2 наведено діаграми компонентів у монолітних та мікросервісних додатках відповідно.

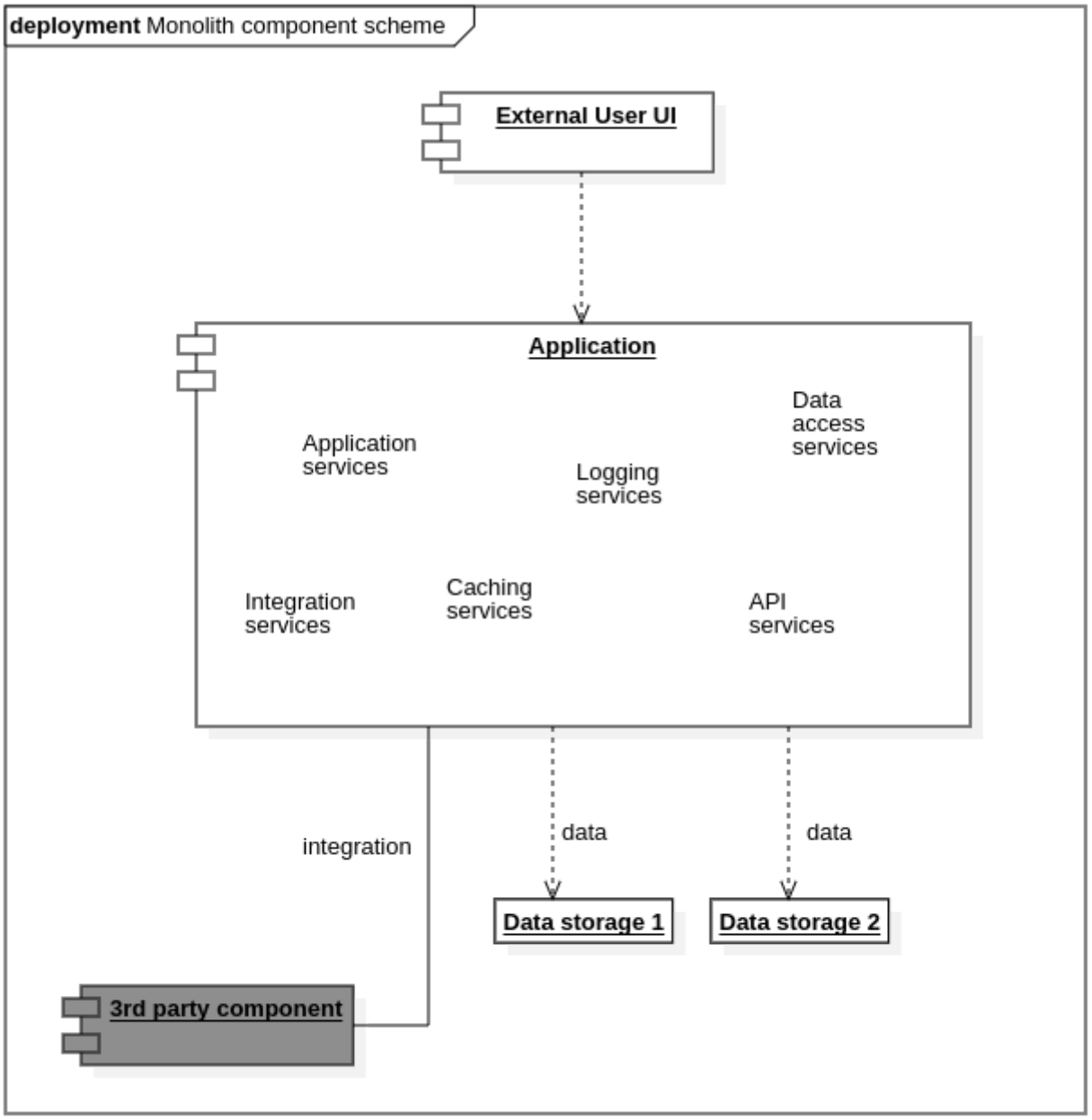


Рисунок 2.1 Діаграма компонентів монолітної системи

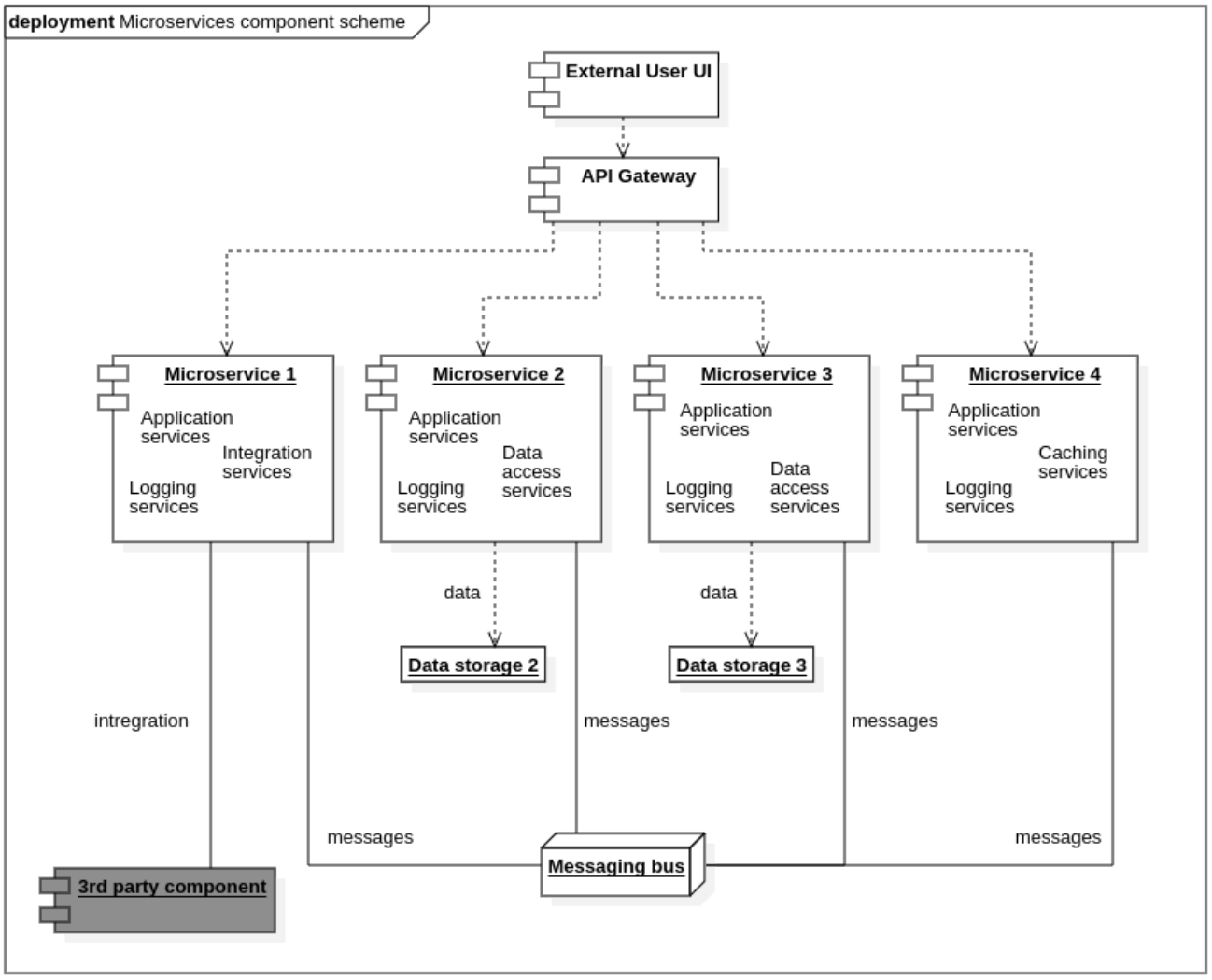


Рисунок 2.2 Діаграма компонентів мікросервісної системи

Не дивлячись на переваги мікросервісної архітектури, вона може бути надмірною, наприклад у невеликих додатках. При початковому проектуванні необхідно оцінювати потенційні навантаження, майбутнє розширення функціоналу, можливі взаємодії зі сторонніми сервісами тощо. Грамотно спроектована система - це запорука простоти підтримки та подальшого розвитку системи в цілому. Саме тому при побудові нової системи обов’язково потрібно оцінити всі потенційні

ризики та обрати найбільш придатну архітектуру. У таблиці 2.1 представлено порівняння монолітної та мікросервісної архітектур з урахуванням переваг та недоліків кожної з них за певними показниками [9].

Таблиця 2.1 Порівняння монолітної та мікросервісної архітектур

Показник	Монолітна архітектура	Мікросервісна архітектура	Причина
Стійкість до навантажень	низька	висока	розподіл навантаження між процесами надає додаткового балансу
Стійкість до відмов	залежить від надійності додатку	висока	відмова єдиної точки входу в обслуговуванні означає повну непрацездатність системи
Простота підтримки	сильно залежить від структурованості компонент	висока	підтримка кожного окремого мікросервісу значно простіше за одночасну підтримку всього додатку
Простота автоматичного тестування	висока	дуже низька	оскільки всі компоненти монолітно додатку знаходяться в одному місці, їх значно простіше тестувати
Простота доставки та розгортання	дуже висока	потребує автоматизації	кожен мікросервіс необхідно розгортати окремо, що накладає додаткові витрати
Здатність до розподіленої розробки	низька	висока	розробкою кожного окремого мікросервісу може займатися окрема команда розробників
Потенційні	залежить від	неявні	зміни у програмному

побічні ефекти	зв’язаності коду		інтерфейсі одного мікросервісу може призвести до помилок у викликаючих мікросервісах
Здатність до горизонтального масштабування	висока	висока	в обох випадках складність створення нових процесів не залежить від кількості початкових процесів
Здатність до вертикального масштабування	низька	висока	монолітний додаток може споживати більше ресурсів, аніж може надати єдиний сервер; мікросервіси можливо розгортати на окремих серверах [10]
Складність реалізації механізмів транзакцій	дуже висока	дуже низька	реалізація розподілених транзакцій тягне за собою великі додаткові зусилля
Додаткові накладення на транспортування повідомлень	відсутні	присутні	мікросервісні додатки потребують встановлення додаткових каналів зв’язку для взаємодії між собою

У світі комерційної розробки існують проблеми з відсутністю чіткого уявлення про те, коли саме необхідно використовувати мікросервіси. Це може спричинити витрати значних інвестицій, як з точки зору зусиль, часу чи грошей, так і з точки зору визначення пріоритетності переходу на мікросервіси над реалізацією нового функціоналу вже наявного монолітного продукту. Крім того, створення фрагментованого додатку ускладнюється тим, що може пройти деякий час, перш ніж команда розробки чи замовник помітять певну вигоду у використанні розподіленої архітектури [9].

2.3 Термінологія основних складових CQRS принципу

Принцип CQRS та його найбільш розповсюджені реалізації оперують сіма основними складовими, що лягають в основу додатку, побудованого за цим принципом.

Плоский клас — це клас, що містить у собі поля та не містить жодного методу, а отже і логіки. Такі класи використовуються для групування та передачі даних з одного прошарку додатку в інший. Наприклад, у мові програмування Java плоскими класами є POJO класи, а у мові програмування C# це POCO класи.

Команда (command) — представляє собою об’єкт, що містить дані про необхідні зміни у сховищі даних. Частіше за все, назва класу такого об’єкту відповідає семантиці дії, що має відбутися. Наприклад, UpdateUserEmailCommand означає, що команда змінює поточну електронну пошту користувача. Клас команди має бути плоским класом.

Запит (query) — представляє собою об’єкт, що містить дані про те, які саме дані необхідно отримати зі сховища. Назва класу таких об’єктів відповідає семантиці як саме будуть отримані дані. Наприклад, UserEmailByIdQuery означає, що необхідно отримати електронну пошту користувача за його унікальним ідентифікатором. Клас запиту має бути плоским класом.

Подія (event) — представляє собою об’єкт, що сповіщає систему про оновлення даних у системі. Назва класу таких об’єктів відповідає семантиці дії, що відбулася. Наприклад, UserEmailUpdatedEvent означає, що електронна пошта користувача була змінена. Події можуть бути створені під час обробки команд або

інших подій та не можуть бути створені під час обробки запитів. Клас події має бути плоским класом.

Обробник (handler) — клас, що містить у собі логіку обробки команди, запиту або події. Обробники запитів завжди мають повертати значення, обробники команд можуть повертати ідентифікатор пов’язаного агрегата бази даних або нічого не повертати. Обробники подій ніколи нічого не повертають, оскільки виконуються у фоні. Програмні класи обробників переважно не є плоскими класами та можуть мати свої залежності.

Перехоплювач (interceptor) — клас, що містить у собі логіку, яка буде виконана та після обробки будь-якої команди чи запиту. Перехоплювачі надають можливість розробникам збирати дані про час та статус виконання кожного окремого обробника, що допомагає знаходити слабкі місця в системі а також відслідковувати помилки в роботі системи. Програмні класи перехоплювачів переважно плоскі, проте не мають на це обмежень.

Валідатор (validator) — клас, що містить у собі логіку валідації даних команди чи запиту. В загальному випадку є різновидом перехоплювача, що дозволяє розробникам перевірити валідність даних всередині контракту до того, як вони досягнуть обробників.

2.4 Аналіз існуючих аналогічних або подібних бібліотек

2.4.1 Бібліотека MediatR

MediatR — бібліотека з відкритим вихідним кодом, що повністю реалізує CQRS принцип та надає можливість розділяти обробку запитів та подій. MediatR дуже легковага, оскільки є повністю автономною, тобто не має залежних бібліотек.

Вільно розповсюджується за допомогою єдиного бібліотечного пакету. Основною перевагами цього пакету є:

— простота — робота всього пакету забезпечена всього 26 класами;
— незалежність — відсутність залежностей дозволяє уникнути конфлікту версій із сусідніми залежностями інших пакетів;
— швидкість — завдяки простоті кодової бази та відсутності додаткових можливостей, на обробку запитів та подій майже не накладається додатковий час опрацювання.

Основною відмінністю бібліотеки MediatR від фреймворку “Standalone” є повна локальність системи, тобто бібліотека не підтримує транспорт та віддалену обробку команд, запитів чи подій.

2.4.2 Бібліотека Kledex

Kledex — ще одна реалізація CQRS принципу, що також реалізує DDD підхід та event sourcing шаблон. Kledex підтримує віддалену обробку запитів, використовуючи одну з двох шин повідомлень: RabbitMQ та Azure ServiceBus. Крім того, дана бібліотека має підтримку декількох сховищ подій.

Основною відмінністю фреймворку “Standalone” від бібліотеки Kledex є повна підтримка локальної та віддаленої обробки команд, запитів та подій, тоді як Kledex має лише підтримку обробки віддалених подій. Крім того, ця бібліотека перевантажена сторонніми можливостями, що є в неї вбудованими та не являються розширеннями, а саме валідація команд та запитів, підтримка окремо синхронної і асинхронної обробки. Окремим недоліком Kledex є перевантаженість базових інтерфейсів, що містять у собі надмірну кількість функціоналу.

2.4.3 Бібліотека LiteCQRS

LiteCQRS — легковага бібліотека, що кардинально відрізняється від попередніх ключовою особливістю — пошук відповідних обробників команд та подій відбувається не за їх типом, а за конвенцією найменувань класів обробників та їх просторів імен. Плюсом такої реалізації є повна свобода від зовнішніх інтерфейсів, що дозволяє не обв’язувати код продукту додатковими залежностями. Ця перевага тягне за собою перелік недоліків:

— низька ефективність коду — такий підхід потребує використання рефлексії, що сильно сповільнює швидкодію програми, а також унеможливлює реалізацію, якщо рефлексія не підтримується мовою програмування;

— низька надійність та підтримуваність коду — будь-яка помилка в найменуванні методу чи класу призведе до неможливості обробити команду чи подію;

— відсутність запитів та їх обробників — отримання даних залишається на розгляд розробника додатку;

— локальність обробників — бібліотека не підтримує транспорт та віддалену обробку команд, запитів та подій.

3. ЗАСОБИ РОЗРОБКИ ПРОГРАМНОГО ПРОДУКТУ

3.1 Мова програмування та цільова платформа

Для розробки фреймворку “Standalone” було обрано мову програмування загального призначення C# версії 8.0. Перша версія мови з’явилася у 2000 році у складі .NET Framework в якості мови прикладного рівня для Common Language Runtime (CLR).

C# є об’єктно-орієнтованою і типобезпечною мовою програмування, що відноситься до сімейства C-подібних мов програмування. Програми, що написані мовою C# забезпечені надійністю та витривалістю завдяки наступним можливостям:

— типобезпека — безпека системи типів мови. C# має сильну статичну типізацію, що виключає можливість появи помилок узгодження типів на стадії виконання. Вихідний код, що містить у собі неузгодженість між операндами команд, не може бути скомпільована, а отже і не може бути запущена;

— автоматичний збір “сміття” — на відміну від C та C++, розробнику не потрібно піклуватися про ручне виділення та звільнення пам’яті. Оточення виконання CLR бере цю задачу на себе, що виключає можливість появи витоку пам’яті;

— узагальнення — дозволяє розробникам писати алгоритми, що можуть працювати з будь-яким типом даних, а отже підвищується рівень перевикористання існуючого коду; крім того, узагальнення гарантують типобезпеку за рахунок перевірки узагальнених типів на етапі компіляції [11];

— потужний механізм обробки виключень — дозволяє розробляти надійні системи, що є стійкими до виключних ситуацій та мають можливість відновлення одразу після вилову виключення;
— крос-платформенність — завдяки платформі .NET Core, програми написані мовою програмування C# можуть бути розгорнуті на операційних системах Windows, Linux та MacOS без жодних змін вихідного коду.

Цільовою платформою фреймворку “Standalone” є .NET Standard 2.1. Це означає, що продукт буде сумісний як із застарівшою версією .NET Framework 4.8, так і з новими редакціями .NET Core 3.0 чи вище [12].

3.2 Середовище розробки JetBrains Rider

JetBrains Rider — це альтернативне інтегроване оточення розробки для мов сімейства .NET. На відміну від найпопулярнішого оточення Visual Studio, Rider є кросплатформним рішенням для .NET розробників, що надає всі необхідні інструменти для комфортної розробки програмного забезпечення не тільки під Windows, але також і Linux та MacOS. З появою .NET Core у 2016 році, ніша оточення розробки під Linux була вільна, тож Rider є просто необхідним інструментом для будь-якого .NET розробника під Unix-системи.

Іншою відмінністю від Visual Studio є інша модель монетизації продукту. JetBrains Rider розповсюджується за місячною чи річною передплатою. Компанія-розробник надає тридцятиденний пробний період, щоб користувачі могли випробувати Rider перед купівлею. Слід відмітити, що JetBrains надає повний пакет своїх продуктів, включаючи Rider, абсолютно безкоштовно для студентів на період навчання.

Rider підтримує .NET Framework, нову платформу .NET Core і проекти на основі Mono. IDE дозволяє розробляти десктопні програми, .NET-сервіси та бібліотеки, розробку ігор під Unity, мобільні додатки Xamarin, а також веб-додатки ASP.NET і ASP.NET Core. Крім того, Rider надає більше 2200 різноманітних інспекцій коду, сотні контекстуальних дій та автогенерацій коду, запозичених з ReSharper, в поєднанні з розширеною функціональністю середовищ розробки на основі платформи IntelliJ [13]. Вбудований відладчик додатків, що націлені на .NET Framework, Mono, а також .NET Core, підтримує покрокове виконання, дозволяє обчислювати вирази поза кодом, відстежувати і змінювати значення змінних. Крім того, Rider включає в себе браузер пакетів NuGet, що дозволяє шукати та встановлювати зовнішні залежності, відслідковувати конфлікти версій, а також консолідувати версії в разі необхідності. Незважаючи на великий набір функцій, Rider — швидка та комфортна для розробки IDE.

Для розробки даної роботи було використано JetBrains Rider останньої версії — 2020.1.2. Основною перевагою останньої версії є використання офіційної версії .NET Core 3.1 як платформи виконання, замість застарілої платформи Mono. Такий перехід дозволяє скоротити навантаження на центральний процесор та оперативну пам’ять на понад 30%.

3.3 Шина обміну повідомленнями RabbitMQ

RabbitMQ — це найпоширеніший брокер повідомлень з відкритим кодом, що реалізує протокол обміну повідомленнями AMPQ. Простіше кажучи, це програмне забезпечення, що дозволяє визначити черги, до яких додатки підключаються для передачі повідомлень. Повідомлення може включати будь-яку інформацію.

Наприклад, вона може містити інформацію про процес або завдання, які повинні починатися в іншій програмі (яка може бути навіть на іншому сервері), або це може бути просто просте текстове повідомлення. RabbitMQ зберігає повідомлення, поки приймаюча програма не приєднається та не зніме повідомлення з черги. Нижче наведена оглядова схема обміну повідомленнями між видавцем та споживачем або обробником повідомлення.

Рисунок 3.1 — Схема обміну повідомленнями між двома додатками через RabbitMQ

RabbitMQ використовує протокол обміну повідомленнями AMQP за замовчуванням. Для початку роботи необхідно встановити бібліотеку, яка реалізує роботу з цим протоколом, наприклад RabbitMQ.Client для C#, php-amqplib для PHP тощо. Клієнтська бібліотека — це інтерфейс прикладного програмування (API) для використання в написанні клієнтських додатків.

Для відладки транспорту повідомлень, RabbitMQ пропонує використовувати веб-інтерфейс адміністратора, що включає в себе повне управління брокером, надає можливість переглядати поточне навантаження на брокер та навіть відслідковувати кожне окреме повідомлення [14].

4. ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ КОМУНІКАЦІЇ МІКРОСЕРВІСІВ

4.1 Структурування рішення

4.1.1 Логічна структура фреймворку

Фреймворк “Standalone” можна поділити семантично на дві частини — абстракції та імплементації. На одну абстрацію припадає одна чи більше імплементацій, що є взаємозамінні. Використання одних компонентів системи іншими відбувається саме через абстракції, що дозволяє замінювати або розширювати функціонал фреймворку, при цьому не змінюючи вже існуючу кодову базу. Такий підхід повністю відповідає останньому з принципів SOLID — Dependency Inversion, згідно якому будь-які програмні модулі вищого рівня не залежать від модулів нижчого рівня, замість цього будь-які модулі залежать лише від абстракцій. Приєднання імплементацій до абстракцій здійснюється за принципом Dependency Injection — створення об’єкту та отримання всіх його залежностей відбувається автоматично за допомогою контейнера залежностей. Такий контейнер конфігурується на початку роботи додатку, а саме відбувається реєстрація реалізацій системи та їх підв’язка під їх абстрації.

4.1.2 Компонентна структура фреймворку

Продукт даної роботи можна поділити на модулі - високорівневі частини рішення, що об'єднують в собі декілька компонентів. Модулі інтегруються між собою за допомогою одного або декількох компонентів. Кожний компонент має у

своїй назві префікс, що відповідає назві свого модулю. Загалом, фреймворк містить у собі чотири модулі: CQRS, DependencyInjection, Exceptional та Tools.

Модуль CQRS містить компоненти, що відповідають за створення, обробку, валідацію та пересилання контрактів:

— Standalone.Cqrs.Abstractions — декларує основні інтерфейси для реалізації так користування принципом CQRS, а саме інтерфейси шин, обробників, перехоплювачів, конвеєрів та контекстів;
— Standalone.Cqrs.Abstractions.Contracts — містить базові інтерфейси контрактів, а саме команд, запитів та подій;
— Standalone.Cqrs — надає основні реалізації конвеєру обробки;
— Standalone.Cqrs.Extensions.FluentValidation - компонент інтеграції з бібліотекою валідації FluentValidation;
— Standalone.Cqrs.Transport.Abstractions — містить інтерфейси для роботи з різноманітними низькорівневими системами обміну повідомленнями;
— Standalone.Cqrs.Transport.RabbitMq.Abstractions — містить абстракції для роботи з брокером повідомлень RabbitMQ;
— Standalone.Cqrs.Transport.RabbitMq — містить реалізацію транспорту повідомлень, що базується на брокері повідомлень RabbitMQ;
— Standalone.Cqrs.Transport.RabbitMq.Autofac — відповідає за інтеграцію з контейнером залежностей Autofac.

Модуль Dependency Injection включає в себе компоненти, що абстрагують роботу з контейнерами залежностей. Це надає можливість інтегрувати фреймворк з будь-яким контейнером залежностей, не зачіпаючи інші модулі. Цей модуль складається з наступних компонентів:

— Standalone.DependencyInjection.Abstractions — декларує загальні інтерфейси адаптерів для роботи з контейнером залежностей;
— Standalone.DependencyInjection.Adapters.Autofac — містить реалізацію адаптера для контейнеру залежностей Autofac.

Модуль Exceptional містить компоненти лише один компонент — Standalone.Exceptional.Abstractions, він містить у собі абстракції для створення програмних виключень та їх метаданих, таких як код та повідомлення.

Модуль Tools включає в себе різноманітні розширення та інструменти, що спрощують розробку продукту. Ці розширення можуть бути використані як всередині фреймворку, так і кінцевим користувачем. Модуль містить лише компонент Standalone.Exceptional.Abstractions, що декларує базові виключення для всього фреймворку.

Взаємодія та залежності пакетів зображені на рисунку 4.1, де білим кольором позначені власні пакети, сірим позначені зовнішні пакети.

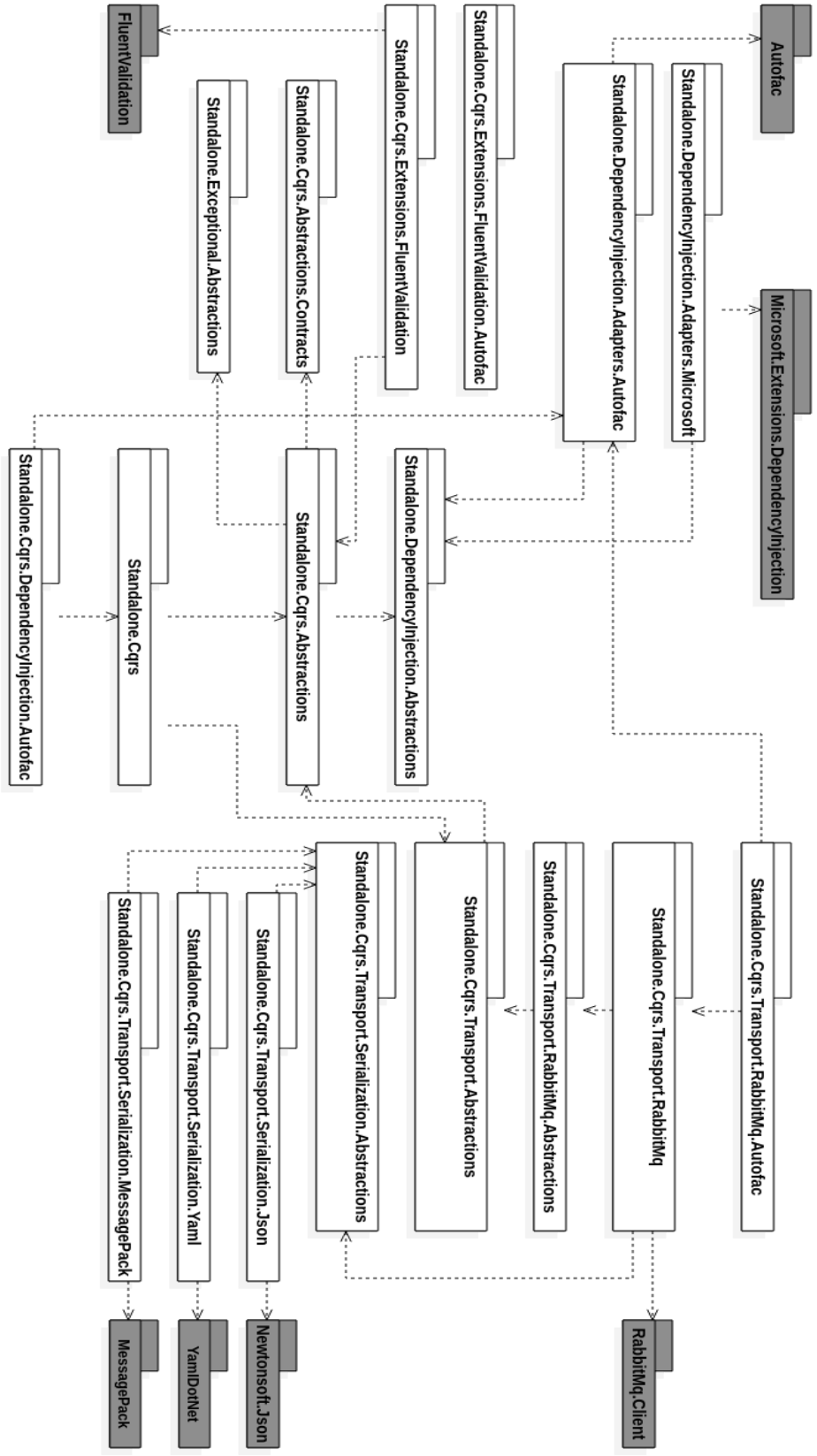

Autofac
Microsoft.Extensions.DependencyInjection
Standalone.DependencyInjection.Adapters.Microsoft
Standalone.DependencyInjection.Adapters.Autofac
FluentValidation
Standalone.Cqrs.Extensions.FluentValidation
Standalone.Cqrs.Extensions.FluentValidation.Autofac
Standalone.Cqrs.Abstractions.Contracts
Standalone.Exceptional.Abstractions
Standalone.Cqrs.Dependencyinjection.Autofac
Standalone.Cqrs
Standalone.Cqrs.Abstractions
Standalone.Dependencyinjection.Abstractions
Standalone.Cqrs.Transport.RabbitMq.Autofac
Standalone.Cqrs.Transport.RabbitMq
Standalone.Cqrs.Transport.RabbitMq.Abstractions
Standalone.Cqrs.Transport.Abstractions
Standalone.Cqrs.Transport.Serialization.Abstractions
Standalone.Cqrs.Transport.Serialization.Json
Standalone.Cqrs.Transport.Serialization.Yaml
Standalone.Cqrs.Transport.Serialization.MessagePack
RabbitMq.Client
Newtonsoft.Json
YamlDotNet
MessagePack

Рисунок 4.1 Діаграма пакетів фреймворку “Standalone”

4.2 Створення та призначення абстракцій

Абстракція, загалом, є фундаментальним поняттям в об'єктно-орієнтованому програмуванні. Процес абстрагування також можна пов’язати з процесом моделюванням, що тісно пов'язаний з перспективою розширюваної архітектури. Розробка абстракцій необхідна для створення точок розширення, що дозволяють додавати новий функціонал без змінення вже існуючої кодової бази. Створені абстракції використовуються для обробки всіх типів методів: команд, запитів та подій. Реалізації приєднуються до абстракцій завдяки контейнеру залежностей. Такі контейнери дозволяють реєструвати залежності за їх інтерфейсами, що використовуються іншими залежностями. Також, контейнери дозволяють реєструвати одразу декілька реалізацій за одним спільним інтерфейсом, що дозволяє розробнику динамічно отримувати всі зареєстровані реалізації та працювати з ними через їх інтерфейси.

Фреймворк “Standalone” представлений одним .NET рішенням (solution). Рішення включає в себе проекти (projects), що можуть використовувати компоненти одне одного, посилаючись на проект, що містить необхідний компонент. Рішення дозволяють групувати проекти у папки, що дозволяє зручно переглядати вихідний код. Такі папки називають модулями рішення. У даній роботи було використано конвенцію найменування, за якою назва проекту наслідує всі назви батьківських папок. На рисунку 4.2 наведена структура рішення “Standalone”.

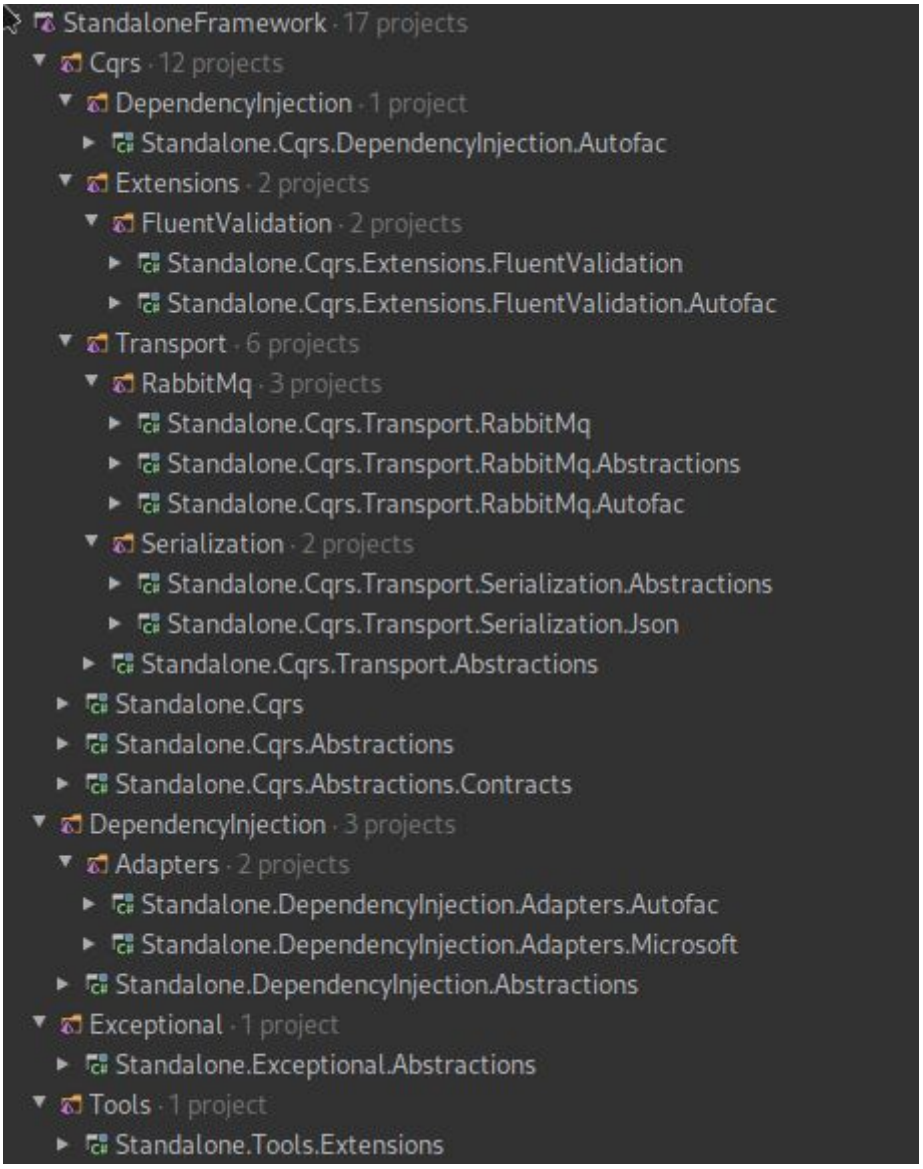


Рисунок 4.2 Структура рішення “Standalone”

Назви директорій рішення (модулів) відповідають за узагальнені ними сфери застосування. Кожен проект, що має постфікс Abstractions у назві, містить абстракції, що відповідають за конкретну сферу їх застосування. Наприклад, абстракції з проекту Standalone.DependencyInjection.Abstractions використовуються у реалізації адаптеру контейнеру залежностей Autofac. В контексті мови

програмування C#, абстракція, зазвичай, це інтерфейс, реалізації якого мають містити властивості та методи, що декларує цей інтерфейс.

Даний продукт містить три основні інтерфейси, з якими взаємодіє розробник додатку — шини передачі: ICommandBus, IQueryBus та IEventBus. Кожна з цих шин відповідає за створення контексту та конвеєру виконання. Для передачі повідомлень та їх обробки необхідно надати контракт. Даний продукт пропонує побудову мікросервісних систем на основі трьох базових контрактів, на основі яких створюються конкретні контракти. Контракт може бути або ідемпотентним, або ні. Ідемпотентність контракту означає, що повторне використання ідентичного контракту не змінює результат виконання цього контракту за умови відсутності паралельного виконання [15]. Розглянемо детальніше кожен контракт.

Команда — контракт виконання дії, що містить у собі контекст виконання цієї дії. За семантикою, команди змінюють поточний стан системи, наприклад оновлення записів у базі даних. Команди мають опціональне повернене значення, але не передбачають семантику запитів. Результатом виконання команди може бути ідентифікатор створеного запису у базі даних додатку або відповідь про обробку даних від зовнішнього сервісу [16]. Обробники команд змінюють стан поточної або зовнішніх систем, про що можливо сповістити решту компонент системи використовуючи контракт події. Оскільки обробники команд змінюють стан системи, контракт команди не є ідемпотентним. У програмній реалізації базовий контракт команди представлений двома інтерфейсами: ICommand та ICommand<TResult> для команд без результату та з результатом відповідно.

Запит — контракт отримання даних, що містить у собі контекст отримання результуючих даних. За семантикою, запити отримують дані, наприклад отримання набору записів з бази даних або читання кешу. Запити передбачають обов’язкове

повертання значення, тому обробники команд завжди повертають значення. Оскільки обробники команд ніколи не змінюють стан системи, контракт запиту є ідемпотентним. У програмній реалізації базовий контракт команди представлений інтерфейсом IQuery.

Подія — контракт сповіщення про зміну стану системи. За семантикою, події сповіщають всі компоненти системи про зміни стану системи. Події виконуються у фоновому режимі, тому обробники подій ніколи не повертають значення. Обробники реагують на зміни стану системи та можуть змінювати пов’язані з подією дані. Крім того, обробники подій можуть створювати інші події. Таким чином можна розбити операцію на ланцюжок подій, що ініціюють одне одного. Варто зазначити, що першу подію завжди створює обробник команд. Контракт події не є ідемпотентним. У програмній реалізації базовий контракт події представлений інтерфейсом IEvent.

Всі наведені вище контракти наслідуються від загального контракту, що завжди має результат і представлений інтерфейсом ISource. Для більшої сумісності процесів обробки різних контрактів, ті з них, що не мають результату, насправді, приховано мають його. Для цього використовується спеціальний тип VoidResult, що не містить у собі даних та служить лише для зведення програмних інтерфейсів до загального виду.

Транспорт — реалізація способу передачі та прийому команд, запитів та подій між мікросервісами. Основною задачею транспорту є серіалізувати повідомлення для його передачі в уніфікований формат та відправити його одному або кільком одержувачам. В контексті принципу CQRS, одержувач являє собою:

— єдиний обробник команди, що опціонально може повертати результат;

— єдиний обробник запиту, що завжди повертає результат;

— множина обробників подій, що ніколи не повертають результат.

Фреймворк “Standalone” поставляє за замовчуванням два транспорти: локальний та RabbitMQ-транспорт. Локальний транспорт доставляє повідомлення всередині процесу, що передає повідомлення у шину, тобто всередині одного мікросервісу. Локальний транспорт не передає дані за межі процесу, а отже необхідність в серіалізації відпадає. В такому випадку повідомлення зберігається в оперативній пам’яті до закінчення роботи конвеєру обробки. RabbitMQ-транспорт використовується для передачі повідомлень між процесами. Така реалізація транспорту передбачає серіалізацію повідомлення та його метаданих, передачу повідомлення, очікування відповіді та її десеріалізацію.

4.3 Реалізація конвеєру обробки команд та подій

Після отримання транспортом повідомлення необхідно почати процес його обробки. Обробка будь-якого повідомлення передбачає декілька етапів. Крім того, необхідно надати розробнику можливість додавати власні етапи обробки. Щоб задовольнити всім вимогам до процесу обробки, необхідно створити інструмент, що дозволяє конфігурувати послідовність викликів вбудованих у фреймворк або користувацьких компонентів. Таку можливість надає реалізований у даній роботі механізм конвеєрної обробки повідомлень. Таке поняття виникло через схожість із звичайним конвеєром — обробка повідомлення виконується не атомарно, а у декілька кроків. Конвеєри у даній роботі представлені у вигляді набору учасників конвеєрної обробки, що виконуються один за одним. У програмному коді конвеєр представлений інтерфейсом IExecutionPipeline та його реалізаціями: CommandExecutionPipeline, QueryExecutionPipeline та EventExecutionPipeline

відповідно до кожного типу контракту. Посередники обробки представлені інтерфейсом IExecutionMiddleware та його імплементаціями. Кожен посередник виконує певну роботу та в залежності від успіху виконання передає контекст виконання наступному посереднику чи ні. Кожен конвеєр будується один раз для кожного контракту, що дає невеликий приріст у швидкості роботи додатку загалом. Різні конвеєри обробки містять різні набори посередників виконання. За замовчуванням, фреймворк містить три посередники, що є загальними для всіх типів контрактів, а також по два специфічні посередники, що використовуються для обробки конкретних типів контрактів. Загальними посередниками є:

— ExceptionHandlingMiddleware — відслідковує виконання інших посередників на предмет виникнення програмних виключень;
— TimeoutMiddleware — встановлює максимальний час обробки контракту. У випадку тайм-ауту виконання, конвеєр обробки завершується з виключенням про тайм-аут;
— InterceptionMiddleware — надає можливість виконувати користувацькі етапи обробки перед та після виклику обробника контракту.

Посередники обробки команд та запитів мають дуже схожу реалізацію, за відмінністю в опціональності результату у команди.

— CommandHandlingMiddleware / QueryHandlingMiddleware — отримує локальну реалізацію відповідного обробника команди / запиту та викликає її. За відсутності такого обробника, обробка делегується наступному посереднику;
— CommandSendingMiddleware / QuerySendingMiddleware — у разі неможливості обробити команду / запит локально, такий контракт відправляється на віддалене виконання. Якщо у контейнері не зареєстровано жодного

транспорту, конвеєр завершується з виключенням про неможливість обробити контракт.

На рисунку 4.3 зображена діаграма класів конвеєру обробки запитів та його складових, що приймають участь в обробці запиту. Діаграми класів для конвеєрів обробки команд та запитів аналогічні.

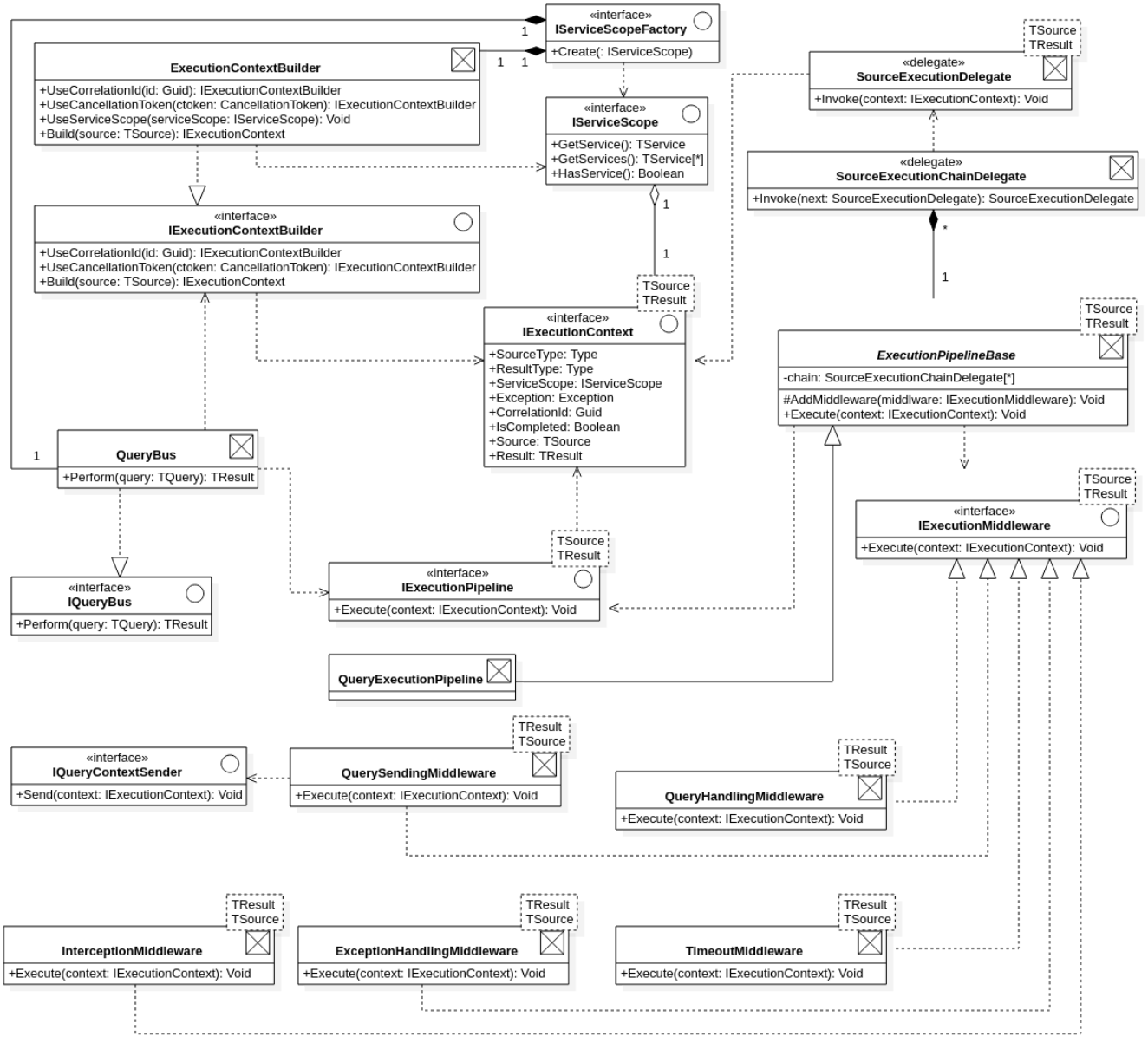


Рисунок 4.3 Діаграма класів конвеєру обробки запитів

Процес обробки подій дещо відрізняється від команд та запитів. По-перше, події завжди виконуються у фоновому режимі, тобто їх обробники не блокують обробку інших контрактів. По-друге, на відміну від команд та запитів, одна подія може мати декілька обробників, що мають бути викликані одне за одним. По-третє, навіть якщо локальна обробка події завершилася виключенням, конвеєр гарантує передачу події у всі зареєстровані транспорти. Конвеєр обробки подій містить аналогічні посередники локальної обробки та публікації події у всі зареєстровані транспорти. Незалежно від успіху роботи посередника локальної обробки, він завжди передає контекс виконання посереднику публікації.

На рисунку 4.4 наведена діаграма послідовності обробки користувацького запиту через налаштований конвеєр обробки запитів.

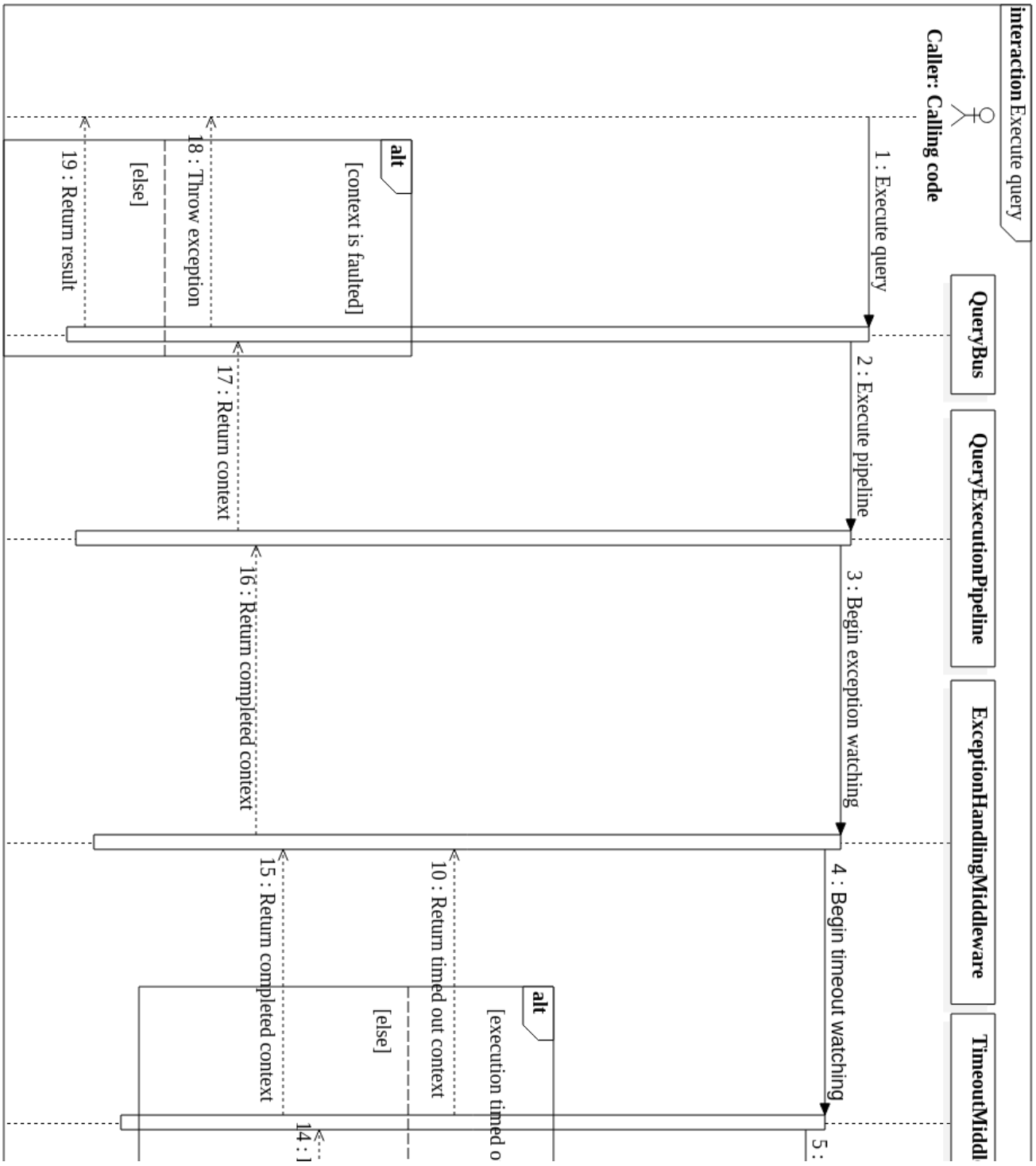


Рисунок 4.4, аркуш 1 Діаграма послідовності конвеєрної обробки запиту

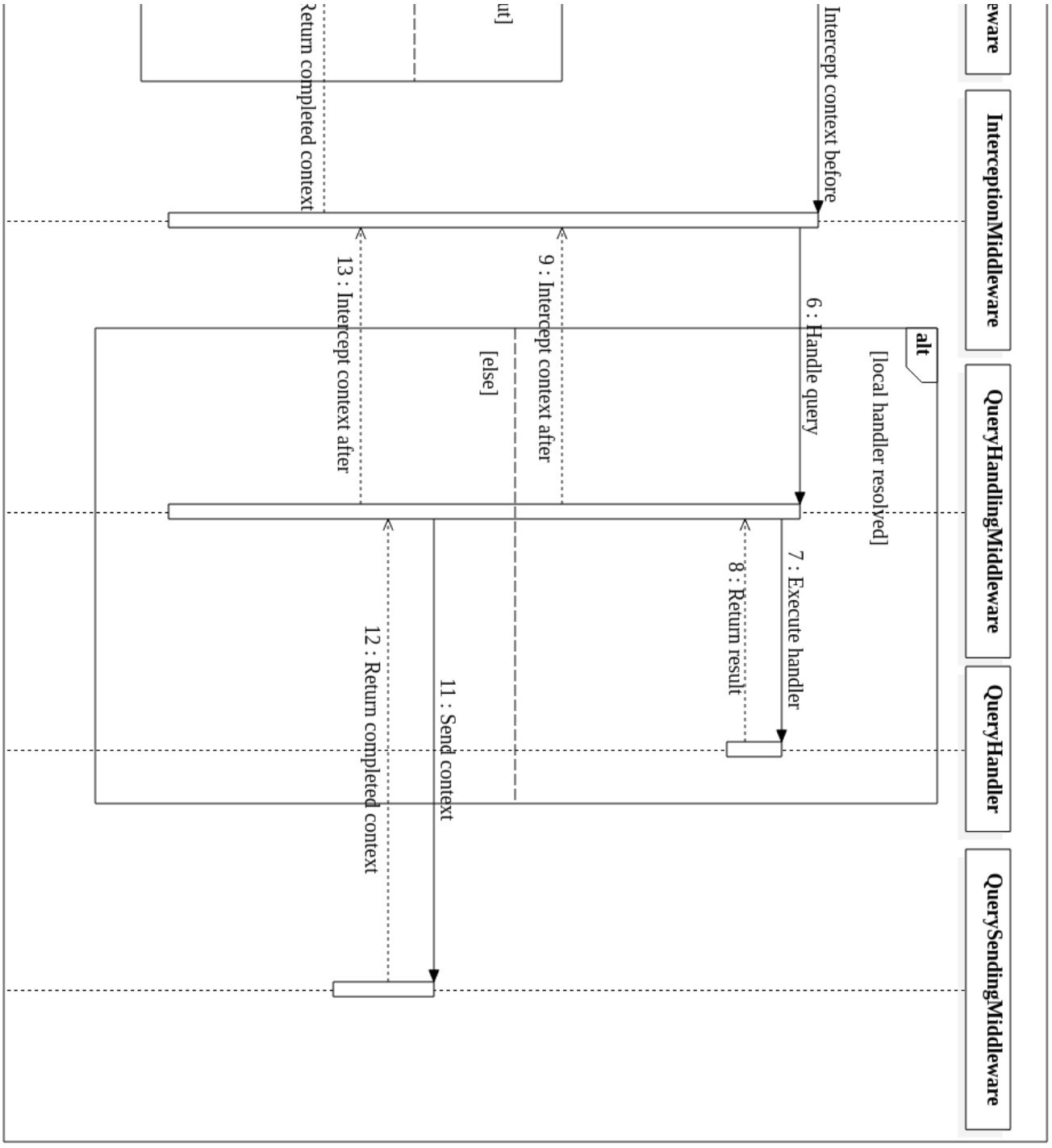


Рисунок 4.4, аркуш 2 Діаграма послідовності конвеєрної обробки запиту

Для кожного базового типу контрактів існує відповідний інтерфейс обробника: ICommandHandler, IQueryHandler та IEventHandler для команд, запитів та подій відповідно. При написанні коду, розробнику необхідно реалізувати конкретний обробник, що реалізує один із заданих інтерфейсів в залежності від типу контракту. Користувацькі обробники реєструються у контейнері залежностей та отримуються у вигляді залежності всередині конвеєра обробки. Аналогічна можливість надається для вбудовування користувацьких етапів обробки. Для цього необхідно реалізувати інтерфейс ISourceInterceptor, реалізації якого мають бути також зареєстровані у контейнері залежностей. Після закінчення циклу обробки в конвеєрі, контекст виконання повертається у шину обробки. Контекст виконання може бути виконано або успішно, або ні. В разі успіху, шина повертає результат виконання, а в разі провалу ініціює виключну ситуацію, що має бути оброблена викликаючим кодом. Виключні ситуації представлені у вигляді спеціальних об’єктів — виключень, що містять у собі дані про неочікувані дані або оточення виконання. У мові програмування C# такі об’єкти представлені наслідниками класу Exception.

4.4 Автоматизація процесу збірки пакетів

Розробка продукту — це довготривалий процес, впродовж якого з’являється новий функціонал, оптимізується вже існуючий та виправляються згодом знайдені помилки. Оскільки продуктом даної роботи являється набір бібліотек, що встановлюються в користувацькі додатки, кінцевий користувач отримує їх за допомогою публічного реєстру пакетів. Кожен пакет представляє собою DLL (Dynamic Load Library) файл, що містить у собі метадані про пакет та наперед скомпільований код. Такі пакети мають свої залежності, що представляються у

вигляді ідентичних пакетів. Кожен пакет має різні версії, що можуть відрізнятися як зовнішніми прикладними програмними інтерфейсами, так і внутрішньою реалізацією цих інтерфейсів. Задля того щоб новий функціонал став доступним кінцевому користувачеві необхідно зібрати змінені пакети та опублікувати їх на загальнодоступному реєстрі пакетів.

Процес збірки та публікації складається з декількох етапів: збірка всього рішення, автоматичне тестування, збірка компонентів у пакети та публікація. Ці етапи можливо виконувати кожен раз вручну, що є достатньо рутинним та накладним по часу процесом. Щоб автоматизувати такі процеси, розробникам пропонується використовувати конвеєри автоматичної збірки, що дозволяють один раз описати всі етапи збірки та виконувати їх за потреби або автоматично за подією в системі. На сьогоднішній день існує безліч таких систем, наприклад werf.io, CircleCI, TravisCI, GitHub Actions тощо. Для автоматизації збірки продукту цієї роботи було обрано GitLabCI конвеєр.

4.4.1 Continuous Integration та Continuous Delivery процеси

Continuous Integration (CI, з англ. — безперервна інтеграція) працює задля безперервної інтеграції та оновлення коду, що розроблений членами команди, у спільному репозиторії. Розробники створюють новий або підтримують існуючий функціонал у персональних гілках, що забезпечує ізольваність одних змін від інших. Після створення такої гілки та внесення в неї необхідних змін, розробник створює запит на злиття змін в основну гілку проекту. В свою чергу, запит автоматично запускає конвеєр для збірки, тестування та перевірки нового коду до об'єднання змін у репозиторій.

Continuous Delivery (CD, з англ. — безперервна доставка) забезпечує доставку результатів збірки коду до пунктів призначення, наприклад виробничі сервери, реєстри пакетів, клієнтські сховища тощо.

Поєднання CI та CD процесів прискорює реалізацію, тестування та доставку результатів команди клієнтам або іншим зацікавленим сторонам. CI допомагає виявляти та зменшувати помилки на початку циклу розробки, а CD автоматизує переміщення автоматично перевіреного коду. CI та CD повинні безперебійно працювати разом, щоб допомогти командам швидко та ефективно будувати додатки, а також мають вирішальне значення для забезпечення повністю автоматизованого процесу розробки [17].

4.4.2 Реалізація конвеєру автоматичної збірки GitLabCI

GitLabCI — технологія автоматизованої конвеєрної збірки, тестування та доставки коду, інтегрована у віддалений менеджер Git-репозиторіїв [18]. Кожний конвеєр цієї системи складається з набору етапів (stages). Етапи виконуються послідовно одне за одним, а кожний наступний етап відштовхується від результатів роботи попереднього. Саме тому такий підхід називають конвеєрним. Етапи складаються з завдань (jobs). Завдання, у найпростішому випадку, представляють собою звичайний скрипт командного рядка, що містить набір команд, які необхідно виконати над вихідним кодом проекту. Завдання, на відміну від етапів, можуть виконуватися незалежно одне від одного, а отже і паралельно. Загалом, конвеєри не обмежуються за кількістю етапів та завдань, що дозволяє розробникам автоматизувати рутинні процеси проектів будь-яких розмірів та складності.

В загальному випадку, конвеєр описується за допомогою одного файлу — .gitlab-ci.yml, що знаходиться у кореневій директорії репозиторію та має

YAML-подібну структуру. Синтаксис цього файлу перевіряється автоматично перед кожним запуском конвеєру, то ж у випадку невірного синтаксису цього файлу конвеєр не буде створено взагалі.

Створення конвеєрів може бути як мануальним, так і автоматичним. Найбільш часто конвеєри налаштовують таким чином, щоб вони були створені одразу зі створенням Git-ярликів (Git-tag). Такі ярлики дозволяють помітити певний стан репозиторію мітками, за якими завжди можна знайти зміни в проекті, що цікавлять розробників. Наприклад, кожний наступний випуск продукту розробники помічають міткою, що містить у назві версію продукту. Кожний такий випуск супроводжується збіркою, тестуванням та публікацією нової версії.

Дана дипломна робота містить реалізацію двох незалежних конвеєрів. Перший з них має назву build/on-merge-request та є повністю автоматичним. Він запускається при створенні будь-якого запиту на злиття та складається лише з одного завдання. Це завдання завантажує всі залежності фреймворку, збирає проект та виконує юніт-тестування. За необхідності, це завдання можна розбити на послідовність завдань, та через невеликі обсяги команд, вони були об’єднані в одне завдання.

Другий конвеєр створюється одразу за створенням Git-ярлику. Конвеєр налаштований таким чином, що можливо збирати кожен окремий модуль фреймворку окремо. Цей конвеєр також складається з одного завдання задля пришвидшення процесу. Завдання містить у собі команди, що заново збирають проект, виконують юніт-тестування, збирають модуль у NuGet-пакети та публікують їх на публічному реєстрі пакетів nuget.org. Наприклад, щоб зібрати рішення та опублікувати його пакети, необхідно лише створити новий Git-ярлик, що має формат release/*X.Y.Z*. На рисунку 4.4 наведено приклад виконання завдання, що збирає рішення та публікує пакети версії 2.0.0 до приватного регістру пакетів.

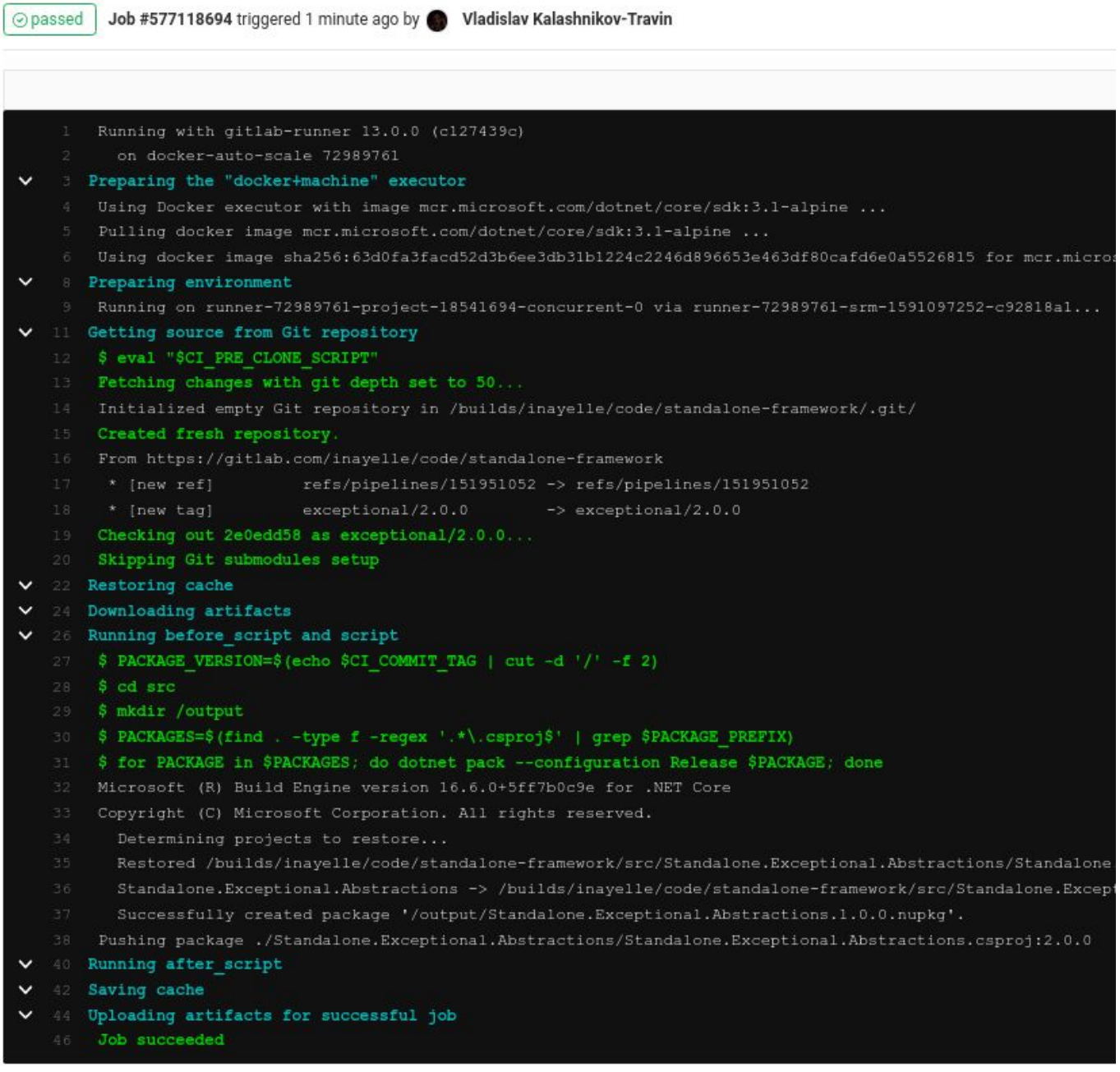


Рисунок 4.5 Приклад виконання завдання GitLabCI

4.4.3 Версіонування пакетів SemVer

У системах з багатьма залежностями випуск нових версій пакету може стати майже нездійсненим завданням. Якщо специфікації залежностей та їх прикладного інтерфейсу не досить виразні, продукту загрожує блокування версій, тобто неможливість оновлення пакетів без необхідності виправляти та перевипускати нові версії кожного залежного пакета.

Рішення цієї проблеми було запропоновано Томом Престоном-Вернером як простий набір правил і вимог, що описують, як слід присвоювати номери версій. Ці правила ґрунтуються на вже існуючих негласних практиках використання версіонування продуктів. Щоб застосувати стандарт версіонування SemVer, необхідно оголосити загальнодоступний програмний інтерфейс продукту, що може складатися з документації продукту або надаватися вихідним кодом. Незалежно від цього, важливо, щоб цей інтерфейс був чітким і точним. Після ідентифікації загальнодоступного програмного інтерфейсу, стає можливим відслідковувати важливість та обсяги змін, ґрунтуючись на змінах лише тільки версії продукту.

Розглянемо запропонований формат версії *X.Y.Z*, де X, Y і Z є мажорною, мінорною та виправляючою версіями. Відштовхуючись від такого формату, можна сформувати наступні правила ведення версій продукту [19]:

— розробник має змінити виправляючу версію продукту, якщо було внесено лише виправлення та взагалі не було змінено програмний інтерфейс;
— розробник має змінити мінорну версію, якщо були внесені зміни в програмний інтерфейс, що не порушують зворотну сумісність з попередньою версією;
— розробник має змінити мажорну версію, якщо було порушено зворотну сумісність з попередньою версією продукту.

У даній роботі систему версіонування SemVer було залучено у процес конвеєрної збірки GitLabCI. Кінцеві користувачі фреймворку можуть відштовхуватися від версій пакетів та оновлювати їх, не хвилюючись за раптове порушення зворотної сумісності з користувацькими продуктами.

4.5 Навантажувальне та стрес тестування

На основі розробленого фреймворку було створено тестовий додаток, що складається з трьох мікросервісів. Кожен з них було налаштовано як на відправку контрактів на віддалену обробку, так і на отримання контрактів та їх локальну обробку.

Створений додаток було протестовано, використовуючи зовнішній засіб тестування k6. Налаштування тестів виконувалось за допомогою JavaScript файлів, що імперативно описують хід тестування. Кожен такий файл можна перевикористати, конфігуруючи його за допомогою JSON файлів конфігурацій. Таким чином, інструмент k6 надає три основні параметри конфігурації: кількість віртуальних користувачів, кількість запитів та тривалість тестування.

Було проведено навантажувальне та стрес-тестування. Навантажувальний тест має показати, як справляється система за нормального навантаження. Обрано наступні параметри тестування:

— тривалість — 10 хвилин
— 250 віртуальних користувачів
— необмежена кількість запитів

Такі параметри відображають середнє навантаження на систему впродовж тривалого часу. Результати навантажувального тесту зображені на рисунку 4.6. За

час тестування, інструментом k6 було зроблено більше 18 мільйонів запитів. За графіком видно, що локальна обробка приблизно в 15 разів швидша, за віддалену. Це зумовлено як брокером повідомлень, так і часовими накладеннями на серіалізацію та десеріалізацію повідомлень, а також на створення віддаленого контексту обробки.

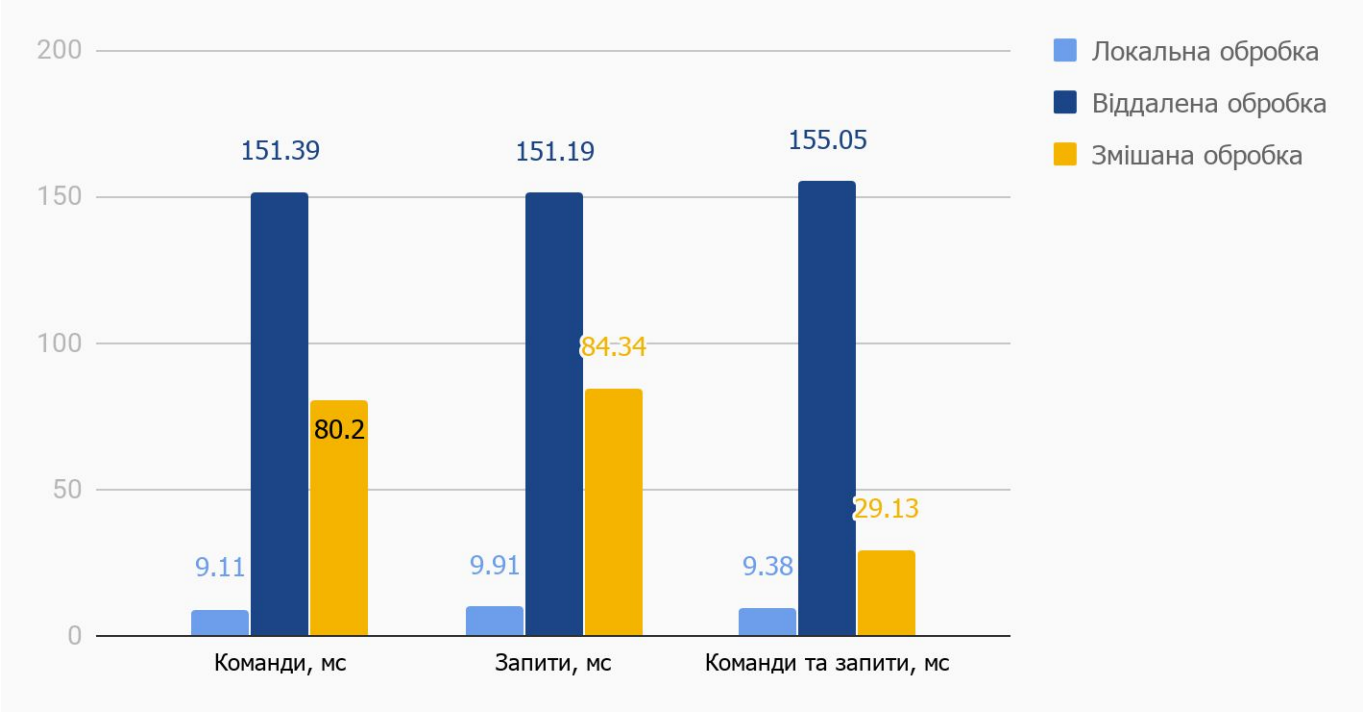


Рисунок 4.6 Графік часу обробки запитів за нормального навантаження

Другим було проведено стрес тестування, що відрізняється більш жорстким характером навантаження, а саме воно є коротшим, але більш щільним. Стрес-тестування призначене для випробування системи за надмірного навантаження. Обрано наступні параметри тестування:

— тривалість — 1 хвилина
— 500 віртуальних користувачів
— необмежена кількість запитів

На рисунку 4.7 зображені результати стрес-тестування.

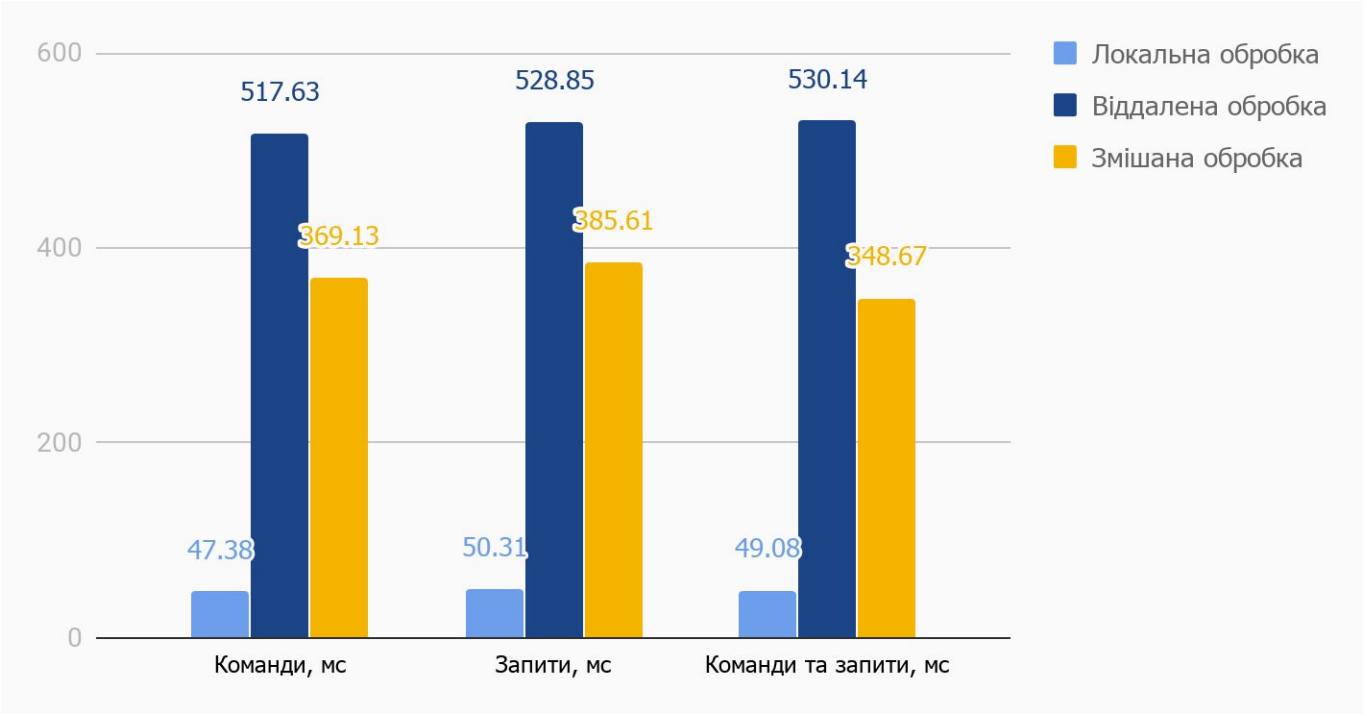


Рисунок 4.7 Графік часу обробки запитів за надмірного навантаження

У порівнянні з попереднім тестом, продуктивність системи впала у 5 разів. Це зумовлено перевантаженими процесором та оперативною пам’яттю системи.

5. ВИКОРИСТАННЯ ФРЕЙМВОРКУ У КОРИСТУВАЦЬКИХ ДОДАТКАХ

5.1 Системні вимоги

Фреймворк “Standalone” поставляється у вигляді NuGet-пакетів, що сумісні з будь-якою мовою сімейства платформи .NET, наприклад C#, F# або VB.NET [20]. Для початку роботи достатньо встановити dotnet-cli у систему розробника. Цільова програмна платформа фреймворку — .NET Standard версії 2.1, тож він повністю сумісний з платформами .NET Core 3.0 та .NET Framework 4.8. Для більшої зручності рекомендується встановити інтегроване середовище розробки Visual Studio версії 16.3 чи вище або JetBrains Rider версії 2020.1 чи вище. Також необхідно впевнитися, що версії залежних пакетів фреймоворку: Autofac, FluentValidation та RabbitMQ.Client — не конфліктують із встановленими розробником пакетами. Для цього необхідно виконати аудит та узгодження версій залежностей.

Розробка додатків, заснованих на базі фреймворку “Standalone” можлива на будь-якій операційній системі сімейств Windows, MacOS та Linux, оскільки .NET Standard є повністю кросплатформною програмною платформою. Для комфортної розробки додатків рекомендовано мати не менше 8 гігабайтів оперативної пам’яті та процесор Intel Core третього покоління чи вище або еквівалентний AMD FX четвертого покоління. Для запуску додатків системні вимоги не визначені, оскільки залежать від об’єму продукту та реального навантаження на систему.

5.2 Встановлення фреймворку “Standalone”

Встановлення фреймворку відбувається через додавання зовнішніх пакетів із загальнодоступного реєстру пакетів nuget.org. Встановлення пакетів відбувається через вбудований в середовище розробки інтерфейс менеджменту пакетів або через командний рядок системи розробника.

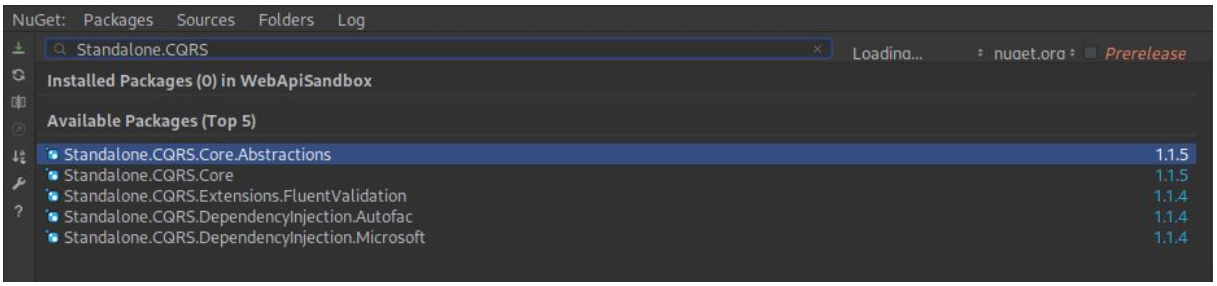


Рисунок 5.1 Встановлення пакетів через інтерфейс середовища JetBrains Rider

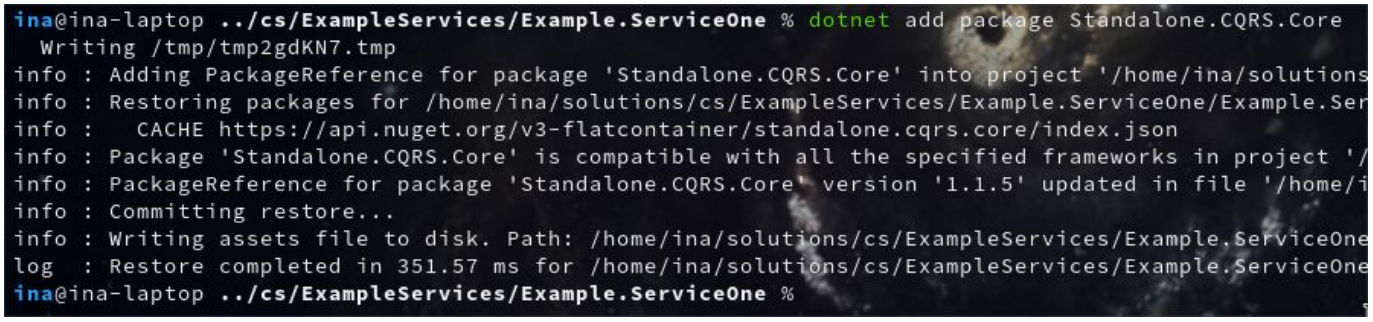


Рисунок 5.2 Встановлення пакетів через інтерфейс командного рядка

Наведені вище способи встановлення ідентичні та відрізняються лише зручністю користування, що ніяк не впливає на загальний функціонал.

Після встановлення необхідно переконатися що всі пакети були успішно додані до проекту. Зробити це можна двома способами: перевірити лог встановлення чи переглянути актуальні залежності проекту розробника використовуючи інтерфейс оточення розробки. На рисунку 5.3 зображено дерево залежностей проекту.

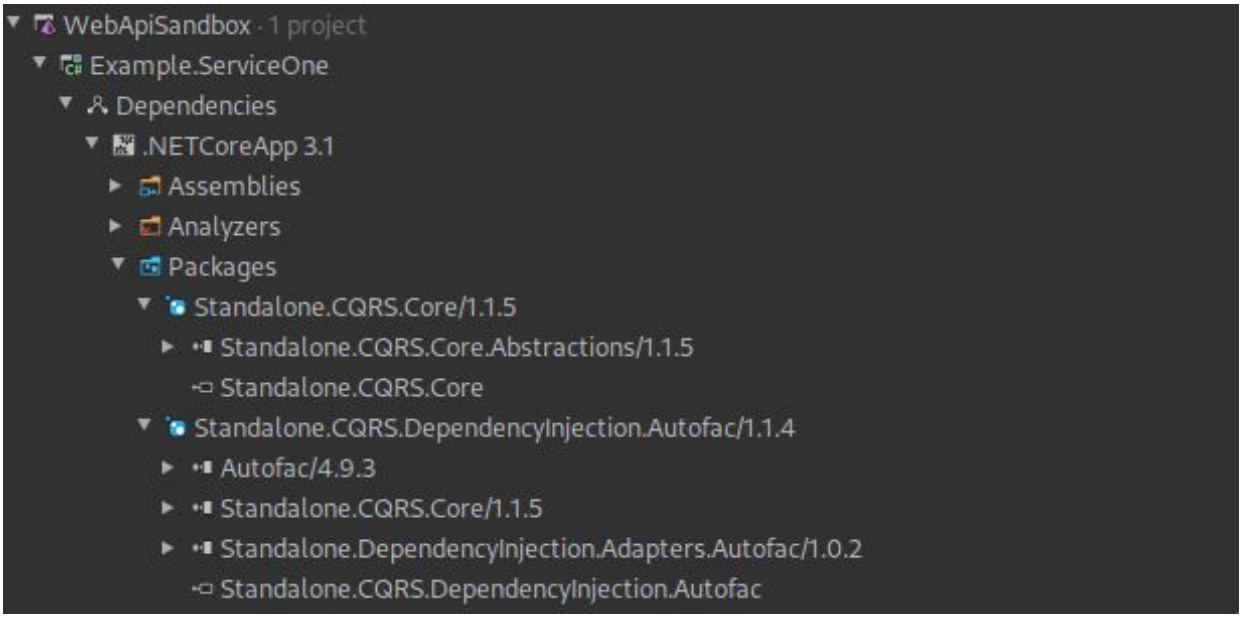


Рисунок 5.3 Перевірка встановлених пакетів у середовищі розробки JetBrains Rider

Після встановлення пакетів, необхідно зареєструвати компоненти фреймворку та його залежності у контейнері залежностей. Крім того, розробник мусить мануально зареєструвати обробники та перехоплювачі контрактів конвеєру обробки.

```
await using var startup = new Startup(builder =>
{
    builder.RegisterStandaloneAdapter();

    builder.RegisterCqrs(options =>
    {
        var assembly = Assembly.GetExecutingAssembly();

        options.RegisterAssemblyHandlers(assembly);
        options.RegisterFluentValidation(assembly);
        options.RegisterInterceptor<StopwatchSourceInterceptor>();

        options.RegisterRabbitMq(rabbit =>
        {
            var serializer = new JsonMessageSerializer();
            var deserializer = new JsonMessageDeserializer();

            var connectionOptions = new RabbitMqTransportOptions(serializer, deserializer)
            {
                Host = "localhost",
                Port = 5672,
                User = "guest",
                Password = "guest",
                VirtualHost = "/"
            };

            rabbit.RegisterCommandSender(connectionOptions);
            rabbit.RegisterQuerySender(connectionOptions);

            rabbit.RegisterEventReceiver(connectionOptions);
        });
    });
});
```

Рисунок 5.4 Приклад реєстрації фреймворку “Standalone” та користувацьких обробників

На рисунку 5.4 зображено приклад реєстрації з використанням контейнеру залежностей Autofac. Мікросервіс з таким налаштуванням контейнеру матиме транспорти для віддаленої обробки команд та запитів, а також буде знаходитись у

режимі очікування зовнішніх подій, тобто тих, що виникатимуть у інших мікросервісах. Всі транспорти налаштовані на використання спільної шини повідомлень RabbitMQ, але за потреби, розробник може налаштувати кожен транспорт під окрему шину. За бажанням, розробник може реалізувати власну реєстрацію під бажаний контейнер залежностей. Варто наголосити, що з користувацькими реалізаціями реєстрації залежностей немає повної гарантії, що конвеєр та транспорт будуть справно функціонувати. Така умовність залежить від кожної конкретної реалізації контейнера, що використовується розробником, а також способу реєстрації та часу життя залежностей у цьому контейнері. Фреймворк рекомендовано використовувати з контейнером залежностей Autofac, що поставляється разом із фреймворком “Standalone” за замовчуванням.

5.3 Рекомендована структура продукту на основі “Standalone”

Оскільки всі додатки, що будуть створені на основі фреймворку “Standalone” матимуть закладену CQRS-архітектуру, кінцевим користувачам рекомендується створювати семантично структуроване дерево директорій, що значно спрощує навігацію по коду користувацькому рішенню під час розробки. Варто відмітити, що на продуктивність додатку це не впливає, оскільки при компіляції всі вихідні коди збираються у файл-збірку. На рисунку 5.5 наведено приклад структури одного з мікросервісів додатку, побудованого за допомогою фреймворку “Standalone”.

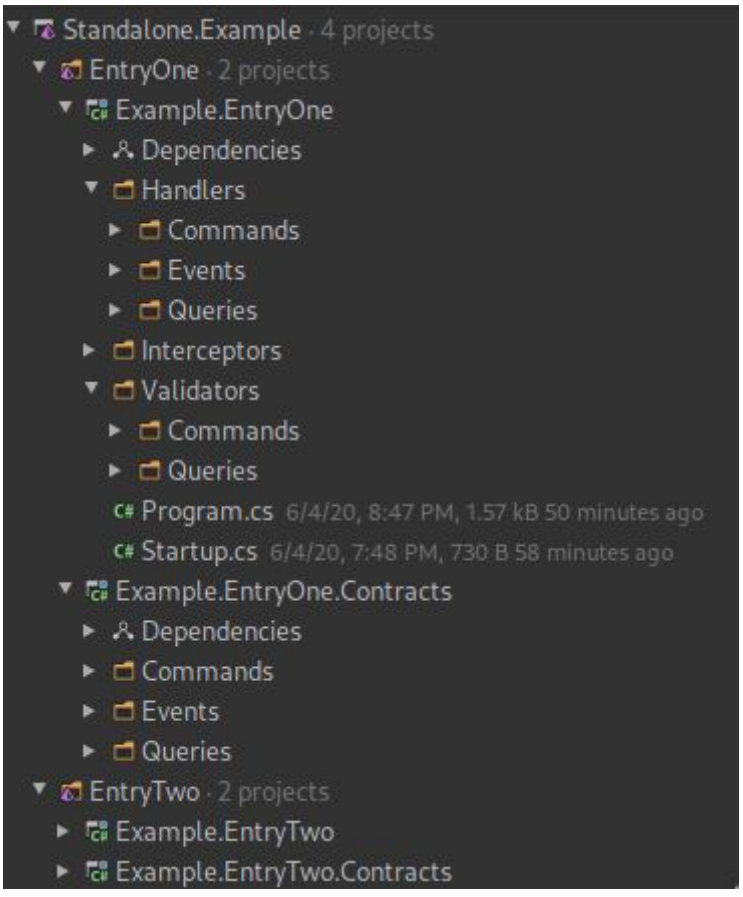


Рисунок 5.5 Рекомендована структура додатку на базі фреймворку “Standalone”

У найпростішому випадку мікросервіс складається за двох проектів:

— Example.EntryOne — точка входу, що містить головний класс, а також всі обробники контрактів, їх валідатори, перехоплювачі та реєстрацію залежностей
— Example.EntryOne.Contracts — проект, що містить лише контракти. Це рекомендується робити задля зменшення розміру збірки з контрактами, адже необхідно буде зробити посилання з іншого мікросервісу на цю збірку

У більш громіздких проектах рекомендується розділяти збірку точки входу від збірки, що містить обробники та валідатори.

5.4 Приклади використання програмних інтерфейсів

Взаємодія кінцевих розробників з інфраструктурою фреймворку виконується через набір програмних інтерфейсів, що надаються компонентом Standalone.Cqrs.Abstractions. Для відправки контрактів на обробку, необхідно отримати реалізації цих шин через використовуваний контейнер залежностей за відповідним інтерфейсом.

Для відправки контракту команди на обробку розробник мусить отримати шину обробки команд та викликати методи Execute, Perform та Publish для команд, запитів та подій відповідно. На рисунку 5.6 зображено приклад виконання команди на створення нового користувача системи.

```
internal sealed class UserCreationController
{
    private readonly ICommandBus _commandBus;
    private readonly SHA512 _sha512;

    public UserCreationController(ICommandBus commandBus)
    {
        _commandBus = commandBus;
        _sha512 = new SHA512Managed();
    }

    public async Task<Guid> CreateUser(CreateUserModel model)
    {
        var command = new CreateUserCommand
        {
            Email = model.Email?.Trim(),
            FullName = model.FullName?.Trim(),
            PasswordHash = _sha512.GetHashString(model.Password)
        };

        var result = await _commandBus.Execute<CreateUserCommand, CreateUserCommandResult>(command);

        return result.UserId;
    }
}
```

Рисунок 5.6 Використання шини обробки команд

Для відправки контракту запиту на обробку розробник мусить отримати шину обробки запитів та викликати метод Perform. На рисунку 5.7 зображено приклад виконання запиту на отримання профілю користувача системи.

```
internal sealed class UserProfileController
{
    private readonly IQueryBus _queryBus;

    public UserProfileController(IQueryBus queryBus)
    {
        _queryBus = queryBus;
    }

    public async Task<UserProfileModel> GetUserProfile(Guid userId)
    {
        var query = new UserProfileByIdQuery
        {
            UserId = userId
        };

        var result = await _queryBus.Perform<UserProfileByIdQuery, UserProfileByIdResponse>(query);

        return new UserProfileModel
        {
            Id = result.Id,
            Email = result.Email,
            FullName = result.FullName
        };
    }
}
```

Рисунок 5.7 Використання шини обробки запитів

Для публікації контракту події на обробку розробник мусить отримати шину обробки подій та викликати метод Publish. На рисунку 5.8 зображено приклад обробника команди та публікації події про створення нового користувача в системі.

```
internal sealed class CreateUserCommandHandler : ICommandHandler<CreateUserCommand, CreateUserCommandResult>
{
    private readonly IUserRepository _userRepository;
    private readonly IEventBus _eventBus;

    public CreateUserCommandHandler(IUserRepository userRepository, IEventBus eventBus)
    {
        _userRepository = userRepository;
        _eventBus = eventBus;
    }

    public async Task<CreateUserCommandResult> Handle(CreateUserCommand source, CancellationToken ctoken)
    {
        var user = new User
        {
            Email = source.Email,
            FullName = source.FullName,
            PasswordHash = source.PasswordHash
        };

        await _userRepository.CreateUser(user);

        var userCreatedEvent = new UserCreatedEvent
        {
            UserId = user.Id
        };

        await _eventBus.Publish(userCreatedEvent, ctoken);

        return new CreateUserCommandResult
        {
            UserId = user.Id
        };
    }
}
```

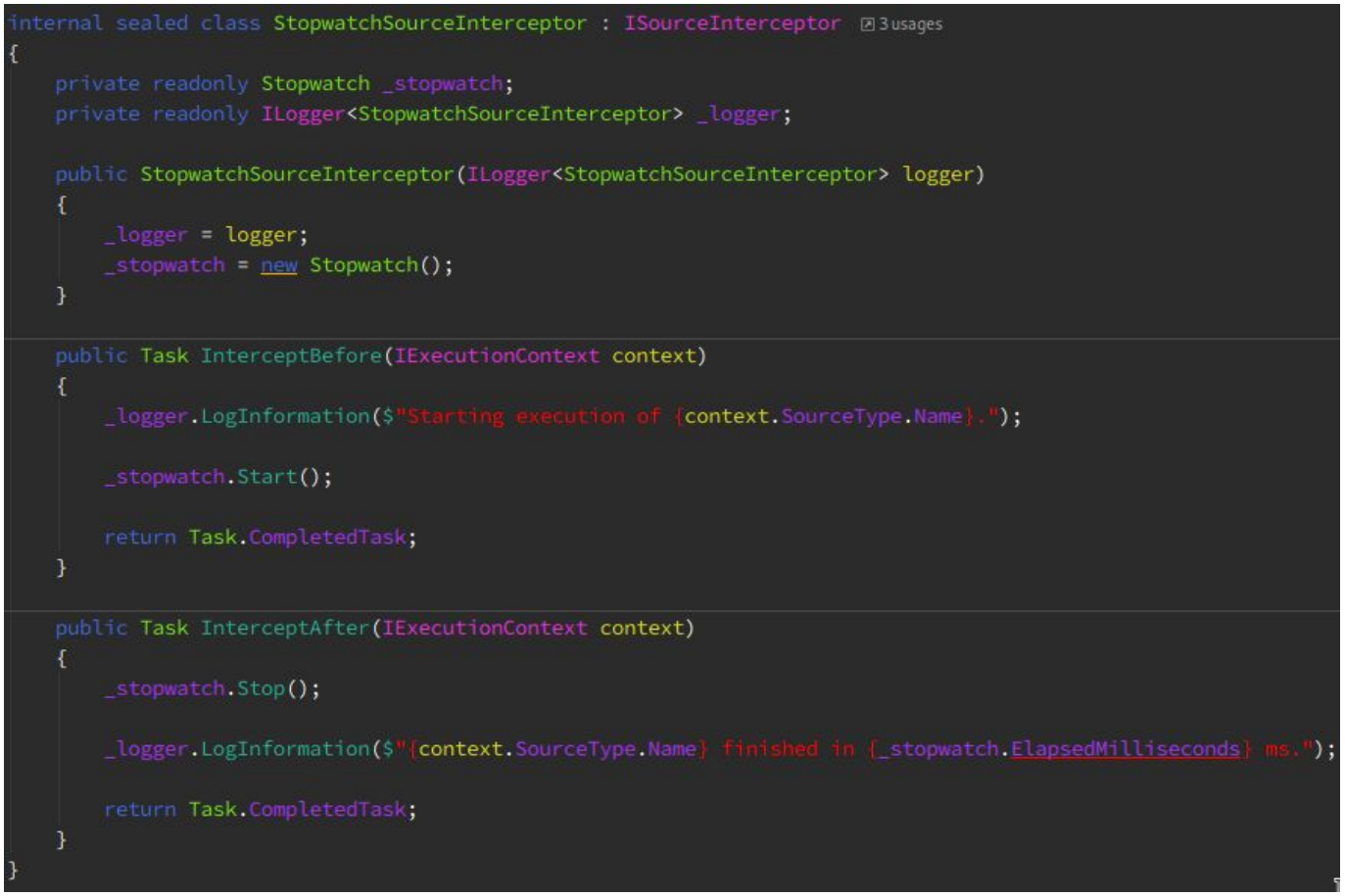
Рисунок 5.8 Приклад використання шини обробки подій

Перед обробкою будь-якого контракту, його необхідно перевірити на валідність. Фреймворк “Standalone” поставляє в своєму складі інтеграцію з бібліотекою валідації даних FluentValidation. На рисунку 5.9 зображено створення валідатора команди на створення нового користувача в системі.

```
internal sealed class CreateUserCommandValidator : AbstractValidator<CreateUserCommand>
{
    public CreateUserCommandValidator(IUserRepository userRepository)
    {
        RuleFor(command => command.Email)
            .NotEmpty()
            .MaximumLength(255)
            .EmailAddress();

        RuleFor(command => command.FullName)
            .NotEmpty()
            .MaximumLength(100);

        RuleFor(command => command.PasswordHash)
            .NotEmpty();

        RuleFor(command => command.Email)
            .MustAsync((email, token) => userRepository.UserWithEmailNotExists(email));
    }
}
```

Рисунок 5.9 Приклад валідатора команди створення нового користувача на базі бібліотеки FluentValidation

Для створення користувацького перехоплювача обробки розробник мусить створити клас, що реалізує програмний інтерфейс ISourceInterceptor та зареєструвати його у контейнері залежностей. На рисунку 5.10 зображено створення перехоплювача, що виконує часовий аудит виконання контрактів команд та запитів.



```
internal sealed class StopwatchSourceInterceptor : ISourceInterceptor  3 usages
{
    private readonly Stopwatch _stopwatch;
    private readonly ILogger<StopwatchSourceInterceptor> _logger;

    public StopwatchSourceInterceptor(ILogger<StopwatchSourceInterceptor> logger)
    {
        _logger = logger;
        _stopwatch = new Stopwatch();
    }

    public Task InterceptBefore(IExecutionContext context)
    {
        _logger.LogInformation($"Starting execution of {context.SourceType.Name}.");

        _stopwatch.Start();

        return Task.CompletedTask;
    }

    public Task InterceptAfter(IExecutionContext context)
    {
        _stopwatch.Stop();

        _logger.LogInformation($"{context.SourceType.Name} finished in {_stopwatch.ElapsedMilliseconds} ms.");

        return Task.CompletedTask;
    }
}
```

Рисунок 5.10 Приклад користувацького перехоплювача, що виконує аудит

Набір програмних інтерфейсів, що надає фреймворк “Standalone” надає безліч варіантів їх використання для розв’язку найрізноманітніших задач. Гнучка система перехоплювачів у поєднанні з роботою з контейнером залежностей надає майже необмежені можливості для розробки бізнес-логіки користувацьких додатків на базі фреймворку, а також розширення чи інтеграції конвеєру обробки з іншими технологіями, що використовуються командою розробки.

ВИСНОВКИ

Відповідно до постановки задачі, у даній роботі було встановлено головні недоліки монолітної архітектури систем та запропоновано альтернативне рішення для створення об’ємних програмних додатків, використовуючи підхід розподілення систем на дрібні компоненти, що взаємодіють між собою шляхом використання спільних контрактів комунікації. Було розглянуто поняття мікросервісної архітектури та кілька її можливих модифікацій. Перед розробкою фреймворку “Standalone”, було проаналізовано три схожі рішення від сторонніх розробників, виявлено їх переваги та недоліки в порівнянні з рішенням, запропонованим у даній роботі. Зазначено основний принцип CQRS, на якому базується реалізація даного програмного продукту, було визначено його основні поняття та компоненти. Зокрема з’ясовано основні вимоги до побудови та підтримки програмних додатків, заснованих на використанні обраного принципу. Було коротко оглянуто поняття брокеру обміну повідомленнями та розглянуто основні положення роботи RabbitMQ. Результатом виконання даної роботи є набір програмних пакетів, на основі яких можливо реалізовувати мікросервісні додатки, використовуючи абсолютно прозорий підхід до обробки запитів, незалежно від місця розташування серед мікросервісів чи мови реалізації того чи іншого обробника команди, запиту чи події. Реалізація фреймворку за замовчуванням інтегрована з контейнером залежностей Autofac та брокером обміну повідомленнями RabbitMQ, проте стандартна поставка не обмежує користувачів у виборі та передбачає надання програмних інтерфейсів для розширення можливостей фреймворку та заміни

брокера повідомлень абсолютно безшовно та без потреби у зміні вже існуючої користувацької кодової бази.

Підсумовуючи результати роботи, загалом було:

1. розглянуто різницю між монолітними та мікросервісними архітектурами додатків;
2. проаналізовано схожі за призначенням фреймворки, виділено їх переваги та недоліки;
3. розроблено загальну архітектуру фреймворку за принципами CQRS та конвеєрної обробки контрактів;
4. закладена підтримка множини контейнерів залежностей;
5. закладена підтримка множини методів серіалізації;
6. закладена підтримка множини брокерів повідомлень;
7. розроблена локальна обробка контрактів;
8. розроблена віддалена обробка контрактів, використовуючи шину обміну повідомленнями RabbitMQ;
9. забезпечено прозоре виконання контрактів, незалежно від місця знаходження обробників;
10. налаштовано автоматичний процес збірки, тестування та публікації компонентів фреймворку.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Newman S. Building Microservices: Designing Fine-Grained Systems / Sam Newman // O'Reilly Media. — 2015. — Vol. 1. — 280 p.
2. Steen M. Distributed Systems / Maarten van Steen, Andrew Tanenbaum // CreateSpace Independent Publishing Platform. — 2017. — Vol. 3.01 — 596 p.
3. Joshi V. Transparency: Illusions of a Single System [Електронний ресурс] / Vaidehi Joshi. — 2019. — Режим доступу: https://medium.com/baseds/transparency-illusions-of-a-single-system-part-1-b01c25f7dddd
4. Victor Y. Common web application architectures [Електронний ресурс] / Youssef Victor, Nick Schonning, Maira Wenzel, Roman Marusyk. — 2019. — Режим доступу: https://docs.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures
5. Torre C. .NET Microservices: Architecture for Containerized .NET Applications / Cesar de la Torre, Bill Wagner, Mike Rousos // Microsoft Developer Division, .NET and Visual Studio product teams. — 2020. — Vol. 1 — 338 p.
6. Fowler M. CommandQuerySeparation [Електронний ресурс] / Martin Fowler. — 2005. — Режим доступу: https://martinfowler.com/bliki/CommandQuerySeparation.html
7. Kumar A. CQRS (Command Query Responsibility Segregation) / Ajay Kumar // Independently published. — 2019. — Vol. 1 — 210 p.

8. Richardson C. Pattern: Command Query Responsibility Segregation (CQRS) [Електронний ресурс] / Chris Richardson. — 2016. — Режим доступу: https://microservices.io/patterns/data/cqrs.html
9. Newman S. Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith / Sam Newman // O'Reilly Media. — 2019. — Vol. 1 — 272 p.
10. Richardson C. Choosing a Microservices Deployment Strategy [Електронний ресурс] / Chris Richardson. — 2016. — Режим доступу: https://www.nginx.com/blog/deploying-microservices/
11. Richter J. CLR Via C# / Jeffrey Richter // Microsoft Press. — 2012. — Vol. 4. — 896 p.
12. Albahari J. C# 8.0 in a Nutshell: The Definitive Reference / Joseph Albahari, Eric Johannsen // O'Reilly Media. — 2020. — Vol. 1 — 1104 p.
13. JetBrains Rider Features [Електронний ресурс] — Режим доступу https://www.jetbrains.com/rider/features/
14. Boschi S. RabbitMQ Cookbook / Sigismondo Boschi, Gabriele Santomaggio // Packt Publishing. — 2013. — Vol. 1. — 290 p.
15. Pacheco V. Microservice Patterns and Best Practices: Explore patterns like CQRS and event sourcing to create scalable, maintainable, and testable microservices / Vinicius Feitosa Pacheco // Packt Publishing. — 2018. — Vol. 1. — 368 p.
16. Richardson C. Microservice patterns with examples in Java / Chris Richardson // Manning Shelter Island. — 2019. — Vol. 1. — 522 p.
17. Lewis J. Microservices [Електронний ресурс] / James Lewis, Martin Fowler. — 2014. — Режим доступу: https://martinfowler.com/articles/microservices.html#InfrastructureAutomation

18. Collins D. GitLab CI/CD Pipeline Configuration Reference [Електронний ресурс] / Dustin Collins, Marin Jankovski. — 2020. — Режим доступу: https://docs.gitlab.com/ee/ci/yaml/
19. Preston-Werner T. Semantic Versioning 2.0.0 [Електронний ресурс] / Tom Preston-Werner. — 2013. — Режим доступу: https://semver.org/lang/uk/
20. Wenzel M. Develop libraries with the .NET Core CLI [Електронний ресурс] / Maira Wenzel, Tom Dykstra, Phillip Carter. — 2017. — Режим доступу: https://docs.microsoft.com/en-us/dotnet/core/tutorials/libraries

ДОДАТОК А

Інструментальні засоби розробки мікросервісів на основі контрактного підходу

СПЕЦИФІКАЦІЯ

УКР.НТУУ”КПІ”.ТВ6133_20Б

Аркушів 2

Київ — 2020

Київ — 2020

Позначення	Найменування	Примітки
Документація		
УКР.НТУУ”КПІ” .ТВ6133_20Б 81-1	Дипломна записка	Зміст дипломної роботи
УКР.НТУУ”КПІ” .ТВ6133_20Б 12-1	Standalone.UML.MetaModel.mdj	UML діаграми
УКР.НТУУ”КПІ” .ТВ6133_20Б 13-1	.gitlab-ci.yml	Налаштування конвеєру GitLabCI
Компоненти		
УКР.НТУУ”КПІ” .ТВ6133_20Б 14-1	Standalone.Cqrs.Abstractions.Contracts.2.0.0.dll	
УКР.НТУУ”КПІ” .ТВ6133_20Б 15-1	Standalone.Cqrs.2.0.0.dll	Реалізація конвеєру обробки контрактів
УКР.НТУУ”КПІ” .ТВ6133_20Б 16-1	Standalone.Cqrs.Transport.RabbitMq.2.0.0.dll	
УКР.НТУУ”КПІ” .ТВ6133_20Б 17-1	Standalone.Cqrs.Transport.Serialization.Json.2.0.0.dll	
УКР.НТУУ”КПІ” .ТВ6133_20Б 18-1	Standalone.Cqrs.Transport.Serialization.Yaml.2.0.0.dll	
УКР.НТУУ”КПІ” .ТВ6133_20Б 19-1	Standalone.Cqrs.Transport.Serialization.MessagePack.2.0.0.dll	

ДОДАТОК Б

Інструментальні засоби розробки мікросервісів на основі контрактного підходу

Текст програмного модулю Standalone.Cqrs

Аркушів 10

Київ — 2020

```
namespace Standalone.Cqrs.Context
{
    internal sealed class ExecutionContext<TSource, TResult> :
IExecutionContext<TSource, TResult>
        where TSource : ISource<TResult>
        where TResult : ISourceResult
    {
        public Guid CorrelationId { get; }

        public bool IsCompleted => IsCompletedSuccessfully |
IsCompletedFaulty;
        public bool IsCompletedSuccessfully => Result != null;
        public bool IsCompletedFaulty => Exception != null;

        public TimeSpan Timeout { get; }
        public TSource Source { get; }
        public TResult Result { get; private set; }
        public Exception Exception { get; private set; }

        public CancellationTokenSource CancellationTokenSource { get;
}
        public IServiceScope ServiceScope { get; }

        public ExecutionContext(TSource source, IServiceScope
serviceScope, IExecutionOptions options,
            CancellationToken ctoken, Guid correlationId)
        {
            CorrelationId = correlationId;
            CancellationTokenSource =
CancellationTokenSource.CreateLinkedTokenSource(ctoken);

            Source = source;
            ServiceScope = serviceScope;
            Timeout = options.ExecutionTimeout;
        }

        public void CompleteWithResult(TResult result)
        {
            Exception = default;
            Result = result;
        }

        public void CompleteWithException(Exception exception)
        {
            Result = default;
            Exception = exception;
```

```
        }

        public void Dispose()
        {
            CancellationTokenSource?.Dispose();
            ServiceScope?.Dispose();
        }
    }
}

namespace Standalone.Cqrs.Context
{
    internal sealed class ExecutionContextBuilder :
IExecutionContextBuilder
    {
        private readonly IServiceScopeFactory _serviceScopeFactory;
        private readonly IExecutionOptions _options;

        private CancellationToken? _cancellationToken;
        private Guid? _correlationId;

        public ExecutionContextBuilder(IServiceScopeFactory
serviceScopeFactory, IExecutionOptions options)
        {
            _serviceScopeFactory = serviceScopeFactory;
            _options = options;
        }

        public IExecutionContextBuilder
UseCancellationToken(CancellationToken ctoken)
        {
            _cancellationToken = ctoken;

            return this;
        }

        public IExecutionContextBuilder UseCorrelationId(Guid
correlationId)
        {
            _correlationId = correlationId;

            return this;
        }

        public IExecutionContext<TSource, TResult> Build<TSource,
TResult>(TSource source)
```

```
            where TSource : ISource<TResult>
            where TResult : ISourceResult
        {
            var scope = _serviceScopeFactory.Create();

            var ctoken = _cancellationToken ?? CancellationToken.None;
            var correlationId = _correlationId ?? Guid.NewGuid();

            return new ExecutionContext<TSource,TResult>(source,
scope, _options, ctoken, correlationId);
        }
    }
}

namespace Standalone.Cqrs.Buses
{
    internal sealed class CommandBus : ICommandBus
    {
        private readonly IExecutionContextBuilder
_executionContextBuilder;

        public CommandBus(IExecutionContextBuilder
executionContextBuilder)
        {
            _executionContextBuilder = executionContextBuilder;
        }

        public Task Execute<TCommand>(TCommand command,
CancellationToken ctoken) where TCommand : ICommand
        {
            return Execute(command, Guid.NewGuid(), ctoken);
        }

        public async Task Execute<TCommand>(TCommand command, Guid
correlationId, CancellationToken ctoken)
            where TCommand : ICommand
        {
            using var context = _executionContextBuilder
               .UseCancellationToken(ctoken)
               .UseCorrelationId(correlationId)
               .Build<TCommand, VoidResult>(command);

            var pipeline =
context.ServiceScope.GetService<IExecutionPipeline<TCommand,
VoidResult>>();
```

```
            await pipeline.Execute(context);

            if (!context.IsCompleted)
            {
                throw new IncompletedExecutionContextException();
            }

            if (context.IsCompletedFaulty)
            {
                throw context.Exception;
            }
        }

        public Task<TResult> Execute<TCommand, TResult>(TCommand
command, CancellationToken ctoken)
            where TCommand : ICommand<TResult>
            where TResult : ICommandResult
        {
            return Execute<TCommand, TResult>(command, Guid.NewGuid(),
ctoken);
        }

        public async Task<TResult> Execute<TCommand, TResult>(TCommand
command, Guid correlationId,
            CancellationToken ctoken)
            where TCommand : ICommand<TResult>
            where TResult : ICommandResult
        {
            using var context = _executionContextBuilder
               .UseCancellationToken(ctoken)
               .UseCorrelationId(correlationId)
               .Build<TCommand, TResult>(command);

            var pipeline =
context.ServiceScope.GetService<IExecutionPipeline<TCommand,
TResult>>();

            await pipeline.Execute(context);

            if (!context.IsCompleted)
            {
                throw new IncompletedExecutionContextException();
            }
            if (context.IsCompletedFaulty)
            {
                throw context.Exception;
```

```
            }

            return context.Result;
        }
    }
}

namespace Standalone.Cqrs.Buses
{
    internal sealed class EventBus : IEventBus
    {
        private readonly IExecutionContextBuilder
_executionContextBuilder;

        public EventBus(IExecutionContextBuilder
executionContextBuilder)
        {
            _executionContextBuilder = executionContextBuilder;
        }

        public Task Publish<TEvent>(TEvent @event, CancellationToken
ctoken) where TEvent : IEvent
        {
            return Publish(@event, Guid.NewGuid(), ctoken);
        }

        public Task Publish<TEvent>(TEvent @event, Guid correlationId,
CancellationToken ctoken) where TEvent : IEvent
        {
            Task.Run(async () =>
            {
                using var context = _executionContextBuilder
                   .UseCorrelationId(correlationId)
                   .UseCancellationToken(ctoken)
                   .Build<TEvent, VoidResult>(@event);

                var pipeline =
context.ServiceScope.GetService<IExecutionPipeline<TEvent,
VoidResult>>();

                await pipeline.Execute(context);

                if (!context.IsCompleted)
                {
                    throw new IncompletedExecutionContextException();
                }
```

```
            }, ctoken);

            return Task.CompletedTask;
        }
    }
}

namespace Standalone.Cqrs.Buses
{
    internal sealed class QueryBus : IQueryBus
    {
        private readonly IExecutionContextBuilder
_executionContextBuilder;

        public QueryBus(IExecutionContextBuilder
executionContextBuilder)
        {
            _executionContextBuilder = executionContextBuilder;
        }

        public Task<TResult> Perform<TQuery, TResult>(TQuery query,
CancellationToken ctoken)
            where TQuery : IQuery<TResult>
            where TResult : IQueryResult
        {
            return Perform<TQuery, TResult>(query, Guid.NewGuid(),
ctoken);
        }

        public async Task<TResult> Perform<TQuery, TResult>(TQuery
query, Guid correlationId, CancellationToken ctoken)
            where TQuery : IQuery<TResult>
            where TResult : IQueryResult
        {
            using var context = _executionContextBuilder
               .UseCancellationToken(ctoken)
               .UseCorrelationId(correlationId)
               .Build<TQuery, TResult>(query);

            var pipeline =
context.ServiceScope.GetService<IExecutionPipeline<TQuery,
TResult>>();

            await pipeline.Execute(context);

            if (!context.IsCompleted)
```

```
            {
                throw new IncompletedExecutionContextException();
            }

            if (context.IsCompletedFaulty)
            {
                throw context.Exception;
            }

            return context.Result;
        }
    }
}

namespace Standalone.Cqrs.Pipelines
{
    internal abstract class ExecutionPipelineBase<TSource, TResult> :
IExecutionPipeline<TSource, TResult>
        where TSource : ISource<TResult>
        where TResult : ISourceResult
    {
        private readonly IList<SourceExecutionChainDelegate<TSource,
TResult>> _chain;

        protected ExecutionPipelineBase()
        {
            _chain = new List<SourceExecutionChainDelegate<TSource,
TResult>>(5);
        }

        public Task Execute(IExecutionContext<TSource, TResult>
context)
        {
            SourceExecutionDelegate<TSource, TResult> chain = ctx =>
Task.CompletedTask;

            for (int index = _chain.Count - 1; index >= 0; --index)
            {
                chain = _chain[index](chain);
            }

            return chain(context);
        }

        protected void AddMiddleware<TMiddleware>(TMiddleware
middleware)
```

```
            where TMiddleware : IExecutionMiddleware<TSource, TResult>
        {
            SourceExecutionChainDelegate<TSource, TResult> @delegate =
next =>
            {
                return async context =>
                {
                    await middleware.Execute(context, next);
                };
            };

            _chain.Add(@delegate);
        }
    }
}

namespace Standalone.Cqrs.Middlewares
{
    internal sealed class ExceptionHandlingMiddleware<TSource,
TResult> : IExecutionMiddleware<TSource, TResult>
        where TSource : ISource<TResult>
        where TResult : ISourceResult
    {
        public async Task Execute(IExecutionContext<TSource, TResult>
context,
            SourceExecutionDelegate<TSource, TResult> next)
        {
            try
            {
                await next(context);
            }
            catch (Exception exception)
            {
                context.CompleteWithException(exception);
            }
        }
    }
}

namespace Standalone.Cqrs.Middlewares.Handling
{
    internal sealed class CommandHandlingMiddleware<TCommand> :
IExecutionMiddleware<TCommand, VoidResult>
        where TCommand : ICommand
    {
        public async Task Execute(IExecutionContext<TCommand,
```

```
VoidResult> context,
            SourceExecutionDelegate<TCommand, VoidResult> next)
        {
            var handler =
context.ServiceScope.GetService<ICommandHandler<TCommand>>(isOptional:
true);

            if (handler == null)
            {
                await next(context);
                return;
            }

            await handler.Handle(context.Source,
context.CancellationToken);

            context.CompleteWithResult(VoidResult.Instance);

            await next(context);
        }
    }

    internal sealed class CommandHandlingMiddleware<TCommand, TResult>
: IExecutionMiddleware<TCommand, TResult>
        where TCommand : ICommand<TResult>
        where TResult : ICommandResult
    {
        public async Task Execute(IExecutionContext<TCommand, TResult>
context,
            SourceExecutionDelegate<TCommand, TResult> next)
        {
            var handler =
context.ServiceScope.GetService<ICommandHandler<TCommand,
TResult>>(isOptional: true);

            if (handler == null)
            {
                await next(context);
                return;
            }

            var result = await handler.Handle(context.Source,
context.CancellationToken);

            context.CompleteWithResult(result);
```

```
            await next(context);
        }
    }
}

namespace Standalone.Cqrs.Middlewares.Handling
{
    internal sealed class EventHandlingMiddleware<TEvent> :
IExecutionMiddleware<TEvent, VoidResult>
        where TEvent : IEvent
    {
        public async Task Execute(IExecutionContext<TEvent,
VoidResult> context,
            SourceExecutionDelegate<TEvent, VoidResult> next)
        {
            var serviceScope = context.ServiceScope;
            var handlers =
serviceScope.GetServices<IEventHandler<TEvent>>();

            var caughtExceptions = new List<Exception>();

            foreach (var handler in handlers)
            {
                try
                {
                    await handler.Handle(context.Source,
context.CancellationToken);
                }
                catch (Exception exception)
                {
                    caughtExceptions.Add(exception);
                }
            }

            await next(context);
        }
    }
}

namespace Standalone.Cqrs.Middlewares.Handling
{
    internal sealed class QueryHandlingMiddleware<TQuery, TResult> :
IExecutionMiddleware<TQuery, TResult>
        where TQuery : IQuery<TResult>
        where TResult : IQueryResult
    {
```

```
        public async Task Execute(IExecutionContext<TQuery, TResult>
context,
            SourceExecutionDelegate<TQuery, TResult> next)
        {
            var serviceScope = context.ServiceScope;

            var handler =
serviceScope.GetService<IQueryHandler<TQuery, TResult>>(isOptional:
true);

            if (handler == null)
            {
                await next(context);
                return;
            }

            var result = await handler.Handle(context.Source,
context.CancellationToken);

            context.CompleteWithResult(result);

            await next(context);
        }
    }
}
```

ДОДАТОК В

Інструментальні засоби розробки мікросервісів на основі контрактного підходу

Опис програмного компоненту Standalone.Cqrs

Аркушів 8

Київ — 2020

АНОТАЦІЯ

Компонент Standalone.Cqrs містить у собі реалізацію конвеєрної обробки команд, запитів та подій. Модуль надає програмний інтерфейс у вигляді інтерфейсів шин ICommandBus, IQueryBus та IEventBus.

Даний модуль виконує наступні задачі:

— створення контексту обробки контрактів;
— виклик користувацьких перехоплювачів;
— локальна обробка контрактів;
— делегування контрактів віддаленим обробникам.

Вхідними даними являються користувацькі контракти, що наслідуються від базових контрактів, наданих фреймворком за замовчуванням. Вихідними даними є результати виконання контрактів.

Як і решта компонентів, розробка цього модулю була виконана в інтегрованому середовищі розробки JetBrains Rider 2020.1.3 мовою програмування C# та націлена на програмну платформу .NET Standard 2.1.

ЗМІСТ

ЗАГАЛЬНІ ВІДОМОСТІ

Програмний компонент Standalone.Cqrs представляє собою NuGet пакет, що встановлюється у користувацькі додатки та служить каркасом для побудови мікросервісної архітектури додатків. Компонент може бути встановлений у будь-який додаток, написаний мовою сімейства .NET та націлений на платформи .NET Core 3.0 чи вище або .NET Framework 4.8 чи вище. Оскільки цільові програмні платформи є кросплатформними, то і сам компонент може працювати на операційних системах Windows, Linux та MacOS.

Єдиною необхідною системною вимогою є наявність встановленого пакету dotnet-cli версії 3.0 або вище.

ФУНКЦІОНАЛЬНЕ ПРИЗНАЧЕННЯ

Компонент Standalone.Cqrs призначений для створення конвеєрної інфраструктури покрокової обробки команд, запитів та подій. Даний компонент бере на себе задачі створення контексту обробки, валідації контракту, виклик перехоплювачів, створення та виклик обробника або, за необхідності, відправки контракту на віддалене виконання.

Даний компонент може бути використаний розробниками, що створюють як монолітну, так і мікросервісну архітектуру додатку.

ОПИС ЛОГІЧНОЇ СТРУКТУРИ

Структурно, компонент Standalone.Cqrs містить реалізації шин, конвеєрів та посередників обробки. Вони реєструються у контейнері залежностей, тому отримати доступ до них можливо лише через їх інтерфейси. Повна структура компоненту зображена на рисунку В.1.

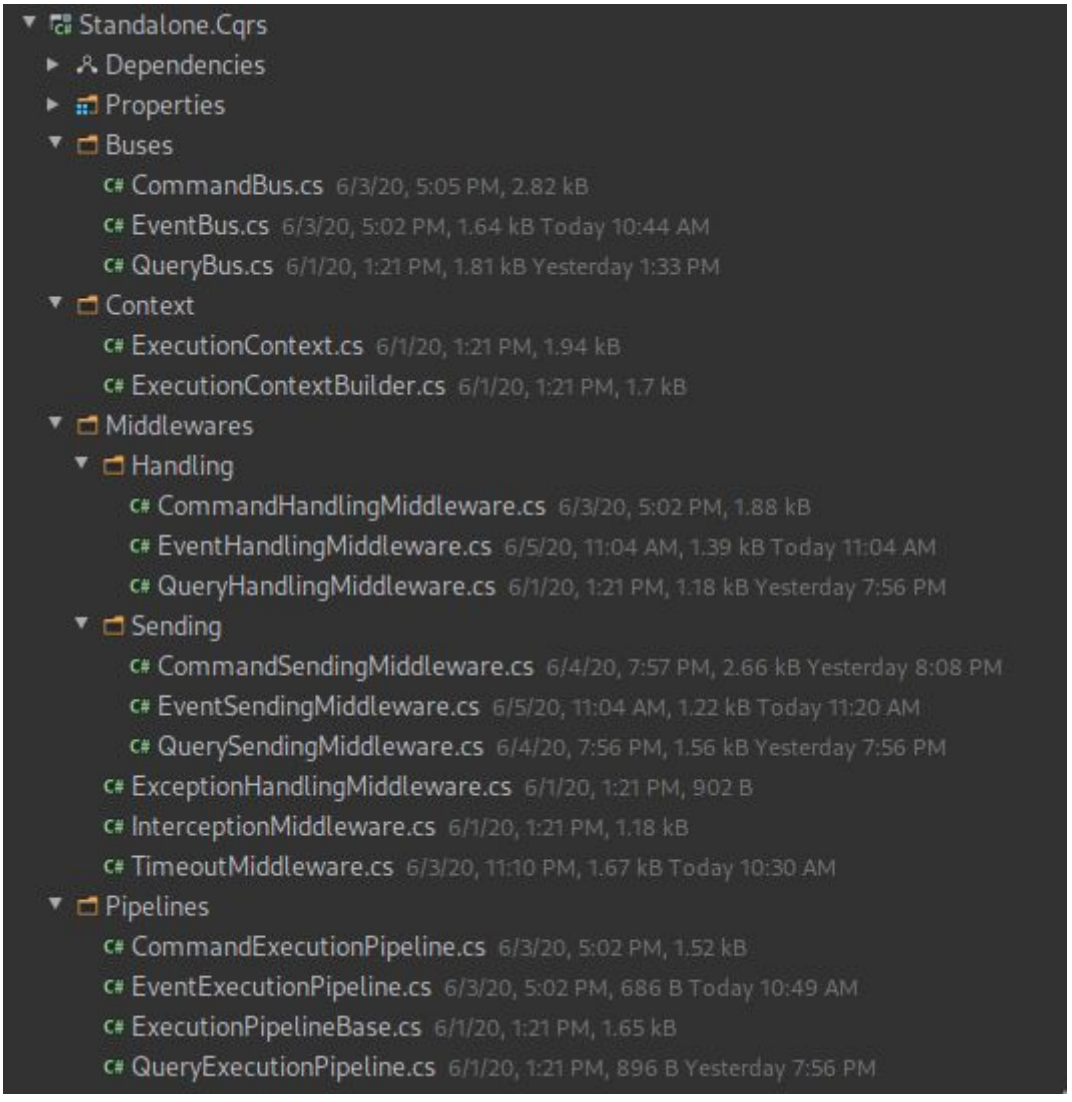


Рисунок В.1 Файлова структура компоненту Standalone.Cqrs

ВИКОРИСТОВУВАНІ ТЕХНІЧНІ ЗАСОБИ

Компонент Standalone.Cqrs, як і всі інші компоненти фреймворку “Standalone”, можуть бути встановлені у будь-який користувацький додаток, що реалізований будь-якою мовою програмування сімейства .NET та націлений на програмну платформу .NET Core 3.0 чи вище або .NET Framework 4.8 чи вище. Для запуску додатків, що засновані на основі фреймворку “Standalone” можна використовувати будь-яке апаратне забезпечення, що керується операційними системами Windows, Linux або MacOS. Крім того, такі користувацькі додатки можуть бути запущені під системами контейнеризації, наприклад Docker/Docker Compose або OpenShift.

ВИКЛИК І ЗАВАНТАЖЕННЯ

Інсталяція компоненту Standalone.Cqrs, як і інших компонентів фреймворку, здійснюється через пакетний менеджер NuGet. Спосіб запуску кінцевого користувацького додатку залежить від типу додатка, мови програмування та операційної системи.

ВХІДНІ ТА ВИХІДНІ ДАНІ

Під час користування компонентом Standalone.Cqrs, вхідними даними є об’єкти користувацьких команд, запитів та подій, реєстрації користувацьких перехоплювачів та обробників контрактів, а також, за необхідністю, налаштування підключення до брокеру повідомлень RabbitMQ.

Користувацькі контракти представляють собою класи, що наслідуються від відповідних базових контрактів ICommand, IQuery та IEvent. Користувацькі перехоплювачі мають реалізовувати інтерфейс ISourceInterceptor. Користувацькі обробники мають реалізовувати інтерфейси ICommandHandler, IQueryHandler або IEventHandler, відповідно до типу контракту.

Користувацькі налаштування з’єднання з шиною повідомлень RabbitMQ встановлюються за допомогою вбудованого класу RabbitMqTransportOptions.

Вихідними даними є результати обробки користувацьких контрактів.

ВІДГУК

керівника дипломної роботи

освітньо-кваліфікаційного рівня „бакалавр”

виконаної на тему : “Інструментальні засоби розробки мікросервісів на основі контрактного підходу”

студентом Калашниковим-Травіним Владиславом Володимировичем

(прізвище, ім’я, по батькові)

Метою даної роботи є вирішення інфраструктурної проблеми комунікації мікросервісів між собою. Результатом роботи став фреймворк “Standalone”, що задає архітектурний каркас для створення мікросервісного додатку на базі архітектурного принципу CQRS та дозволяє розробникам сконцентруватися на розв’язанні задач бізнес-логіки і не зважати на інфраструктурні проблеми комунікації мікросервісів між собою. Основоположним для побудови розподілених систем на базі фреймворку ста контрактний підхід, за яким мікросервіси мають спільні визначення контрактів, за допомогою яких відбувається звернення одного мікросервісу до іншого. Такий підхід позитивно впливає на типобезпеку створення та обміну повідомленнями, а також задає чіткий програмний інтерфейс для кожного мікросервісу.

Зважаючи на вищезазначене, вважаю, що Калашникову-Травіну В.В. може бути присвоєна кваліфікація бакалавра з інженерії програмного забезпечення з напряму підготовки 121 Інженерія програмного забезпечення за спеціалізацією: Програмне забезпечення розподілених систем.

Керівник дипломної роботи

доцент, канд. ф.-м. наук _____ Тарнавський Ю. А.

(посада, вчені звання, ступінь) (підпис) (ініціали, прізвище)

РЕЦЕНЗІЯ

на дипломну роботу

освітньо-кваліфікаційного рівня “бакалавр”

виконаної на тему : “ Інструментальні засоби розробки мікросервісів на основі контрактного підходу”

студентом Калашниковим-Травіним Владиславом Володимировичем

(прізвище, ім’я, по батькові)

Представлена на рецензію дипломна робота включає пояснювальну записку, презентацію доповіді, доповідь та розроблене програмне забезпечення. Всі матеріали відповідають затвердженій темі та завданню на дипломне проектування.

Актуальність теми дипломної роботи полягає в тому, що з ростом складності програмних систем необхідна принципово нова організація архітектури програмних продуктів. Існує потреба в заміні застарілої та мало ефективної монолітної архітектури додатків, яка значно сповільнює розвиток та ускладнює розширення громіздких програмних комплексів, що в свою чергу підвищує вартість розробки та складність у підтримці таких комплексів. При цьому, альтернатива не повинна ускладнювати розробку в плані турбування про реалізацію комунікації компонентів систем, а навпаки, надавати розробникам прозоре рішення для побудови мікросервісного додатку.

Дипломна робота виконана на високому професійному рівні, відповідає вимогам, висунутим в технічному завданні, і заслуговує оцінки “відмінно”.

Зважаючи на вищезазначене, вважаю, що Калашникову-Травіну В. В. може бути присвоєна кваліфікація бакалавра з інженерії програмного забезпечення з напряму підготовки 121 Інженерія програмного забезпечення за спеціалізацією: Програмне забезпечення розподілених систем.

Рецензент

Доцент кафедри АЕС і ІТФ,
к.т.н, доцент — Є.В.Новаківський
(посада, вчені звання, ступінь) (ініціали, прізвище)